# L'Expérimentation d'algorithmes distribués sur machines parallèles avec Echidna

Jean-Marc Jézéquel, Claude Jard

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1528

*Programme 1*
*Architectures parallèles, Bases de données,*
*Réseaux et Systèmes distribués*

# L'EXPÉRIMENTATION D'ALGORITHMES DISTRIBUÉS SUR MACHINES PARALLÈLES AVEC *ECHIDNA*

**Jean-Marc JEZEQUEL**
**Claude JARD**

**Septembre 1991**

*RR-1528*

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex . UNIRISA 950 473 F
Télécopie : 99 38 38 32

# L'expérimentation d'algorithmes distribués sur machines parallèles avec *ECHIDNA*

Jean-Marc JÉZÉQUEL
Claude JARD
E-mail: jezequel@irisa.fr ,jard@irisa.fr

**Publication Interne n° 603 - Septembre 1991, 64 pages.**

**Résumé**

Nous présentons dans ce document une introduction aux techniques de spécifications formelles, de validation et de prototypage sur systèmes distribués. Le premier chapitre est une présentation générale des bases sémantiques et de l'utilisation de techniques de descriptions formelles pour aider à concevoir, spécifier, valider et expérimenter des protocoles de communication. Le second chapitre présente dans le détail la technique de description formelle normalisée appelée Estelle, qui a été choisie comme langage d'entrée pour *ECHIDNA*. Le troisième chapitre constitue une sorte de manuel de référence de *ECHIDNA*, et montre comment l'utiliser pour faire de l'expérimentation d'algorithmes distribués sur machines parallèles.

**Programme 1**

# Experimenting Distributed Algorithms on Parallel Machines with *ECHIDNA*

**Abstract**

This document presents an introduction to formal specifications techniques, validation and prototyping on distributed systems. The first chapter is a general presentation of the semantic basis of formal specifications and their use to design, validate and prototype communication protocols. The second chapter presents in some details a normalized formal description technique called Estelle, which is used in the *ECHIDNA* tool. The third chapter is an *ECHIDNA* reference manual, and shows how one can use it to prototype distributed algorithms on distributed systems.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE (U.R.A 227) UNIVERSITE DE RENNES I - I.N.S.A. DE RENNES
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE (UNITE DE RECHERCHE DE RENNES)

## Remerciements

*Nous tenons à remercier pour leur contribution au développement de* ECHIDNA, *Claire Dichl (interface X11/Motif et T-Node/Helios), Frédéric Guidec (T-Node/Helios), Rémi Guetschel (interface Suntools et ébauche de ce manuel), Thierry Jéron (T-Node/3LC), Noël Plouzeau (Interface X11/Motif) ; et tous ceux qui ont eu le redoutable privilège d'être les premiers utilisateurs de* ECHIDNA.

# Table des matières

# Table des figures

# Introduction

Au cœur des réseaux informatiques interviennent les protocoles de communication, qui sont des règles de dialogue permettant l'échange d'informations entre des entités distantes. De par la nature même d'un réseau de télécommunication, on est confronté aux problèmes suivants :

- un ordinateur n'a pas une vision globale instantanée du réseau,
- les lignes de transmission ne sont pas toujours fiables,
- les matériels reliés peuvent être hétérogènes.

Ceci implique qu'un protocole est un objet complexe et difficile à appréhender intellectuellement compte tenu du parallélisme. Construire, mettre en œuvre et prouver l'exactitude d'un protocole sont autant de tâches délicates. Aussi, on est amené à utiliser des outils de validation (vérification exhaustive, simulation) permettant de s'assurer qu'un protocole est conforme au service demandé.

L'introduction massive du parallélisme dans les ordinateurs modernes relève de la même problématique ; et l'algorithmique distribuée, située au carrefour de ces préoccupations, est le point où les communautés scientifiques des télécommunications et de l'informatique répartie se rejoignent. En effet, que ce soit pour des réseaux à grande distance, des réseaux locaux où des machines massivement parallèles faiblement couplées, le modèle sous-jacent est celui du système réparti (ou distribué), c'est à dire un ensemble de sites (processeurs ou machines) communiquant seulement par échanges de messages au travers d'un réseau de communication point à point. Selon le point de vue où l'on se place, les *programmes* qu'on écrit pour ces systèmes répartis sont des algorithmes distribués ou bien des protocoles ; dans la suite, nous considérerons que ces termes sont équivalents.

Le projet *ECHIDNA* a pour ambition de contribuer à l'effort entrepris depuis quelques années pour comprendre et utiliser les systèmes répartis, en particulier en France par le groupement $C^3$ (Coopération, Concurrence et Communication) suscité par le CNRS. *ECHIDNA* a pour but de faciliter la maîtrise de la programmation de ce type de machines, en fournissant des outils pour aider à concevoir et mettre au point leurs programmes, en observer les comportements et en mesurer les performances.

Avec le choix du langage Estelle (normalisé par l'ISO) pour la description formelle d'algorithmes distribués, *ECHIDNA* s'inscrit dans la continuité d'outils de validation comme Xésar et surtout Véda, qui a d'ailleurs servi pour son prototypage.

Dans l'angle du prototypage, le domaine d'application d'*ECHIDNA* s'étend donc de la validation de protocoles à la programmation de machines parallèles, en passant par

l'enseignement du parallélisme ou la construction d'environnements de programmation répartis.

**Guide de lecture** Le premier chapitre de ce document est une présentation générale des bases sémantiques et de l'utilisation de techniques de descriptions formelles pour aider à concevoir, spécifier, valider et expérimenter des protocoles de communication.

Le second chapitre présente dans le détail la technique de description formelle normalisée appelée Estelle, qui a été choisie comme langage d'entrée pour *FCHIDNA*.

Le troisième chapitre constitue une sorte de manuel de référence du *FCHIDNA*, et montre comment l'utiliser pour faire de l'expérimentation d'algorithmes distribués sur machines parallèles.

# Chapitre 1

# Les techniques de description formelle de protocoles

## 1.1 Nécessité et but des descriptions formelles de protocole

### 1.1.1 Les protocoles de communication

Les protocoles interviennent dans les réseaux informatiques. Un réseau est composé d'un ensemble d'entités, séparées géographiquement, qui communiquent entre elles en s'échangeant des messages. L'ensemble de ces échanges est guidé par un protocole représentant les règles de dialogue et les procédures suivies. Le protocole vu dans sa globalité est souvent divisé en une série de protocoles ; ce découpage rendant plus facile la compréhension et la maintenance du protocole global. Par exemple, on pourrait avoir un protocole pour l'établissement de la connexion, un deuxième pour l'échange effectif des informations et un dernier pour terminer la communication. Cette structuration des protocoles prend tout son sens dans la représentation des sept couches du modèle OSI [19].

### 1.1.2 Les problèmes liés aux protocoles de communication

Comme les lignes qui relient les ordinateurs ne sont pas toujours fiables, on crée des protocoles chargés de corriger les erreurs ; mais ces protocoles introduisent en général d'autre type d'erreurs (perte, duplication ou déséquencement des messages). La conception de ce type de protocoles peut être assez complexe, et la difficulté d'appréhender dans son ensemble un réseau de télécommunication mettant en jeu plusieurs processus parallèles (les ordinateurs n'ont pas une vision globale du réseau), ainsi que l'hétérogénéité des matériels mis en jeu ne facilitent pas les choses.

Voici quelques exemples des différents types d'erreurs les plus fréquentes :

- l'interblocage (deadlock) qui arrive lorsque chacune des entités est en attente d'un signal de réponse de l'autre ; le système est bloqué et ne peut plus progresser,
- la réception non spécifiée, lorsqu'on oublie d'associer un traitement à un message dans un certain état,

- le problème du bouclage survient lorsque les entités sont bloquées dans l'échange de la même séquence de messages,
- il peut y avoir duplication ou perte des messages lors de problèmes de retransmission.

### 1.1.3 Nécessité d'une méthode d'analyse

Au vu de ces problèmes, liés à l'importance des applications et à la complexité des systèmes distribués, on sent la nécessité d'avoir une méthode d'analyse et de conception efficace pour la production de logiciel. On aimerait pouvoir décrire de manière claire et lisible un protocole dans un formalisme approprié en évitant l'usage de la langue naturelle, par trop imprécise, et qui peut conduire à des ambiguïtés d'interprétation. Cette méthode d'analyse doit être capable de répondre à la question : *Comment un protocole fonctionne-t-il dans telle condition précise ?* Elle doit aussi aider à mieux comprendre le protocole.

Aussi, une telle méthodologie de conception doit vérifier un certain nombre de critères :

**pouvoir d'expression :** Une technique de description formelle doit permettre de spécifier les protocoles et les services des sept couches du modèle ISO,

**bonne définition :** Une technique de description formelle doit être fondée sur un modèle mathématique formel qui permette la validation des spécifications,

**structuration :** Une technique de description formelle doit fournir le moyen de structurer les spécifications afin qu'elles soient compréhensibles et faciles à maintenir,

**abstraction :** Une technique de description formelle doit permettre de représenter un protocole à des niveaux d'abstractions différents, du concept général à la description détaillée, et doit essayer d'être indépendante des aspects d'implantation.

### 1.1.4 De la spécification à l'exploitation d'un protocole

Une méthodologie de développement d'un protocole de communication consisterait à suivre les étapes suivantes :

- description formelle,
- validation du protocole,
- implantation du protocole conformément à sa description,
- tests de conformité et évaluation de performances.

Dans la suite de ce chapitre, nous allons développer les deux premières étapes de cette démarche.

## 1.2 Les techniques de description formelle

### 1.2.1 Introduction

Dans une technique de description on peut distinguer trois composantes :

- un langage de description,

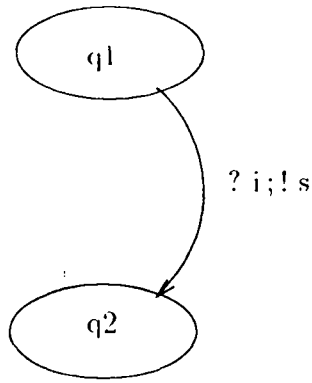


Figure 1.1 : Représentation graphique d'un automate d'états finis

- un modèle mathématique,
- et une relation qui, à chaque programme écrit dans le langage, associe son interprétation dans le modèle mathématique.

Le modèle mathématique généralement utilisé est le *Système de Transition Étiqueté*. C'est un graphe où les noeuds (représentant les états possibles du système) sont reliés entre eux par des arcs étiquetés par les actions qui sont exécutées en passant d'un noeud à un autre.

Nous pouvons distinguer deux types de techniques : d'une part les techniques de type opérationnel qui permettent de construire le *Système de Transition Étiqueté*, et d'autre part les techniques de type logique qui permettent d'énoncer des propriétés qui devront être satisfaites par le *Système de Transition Étiqueté*.

Les techniques de type opérationnel sont caractérisées par le fait qu'il existe un traducteur qui associe à tout programme écrit dans le langage de description un *Système de Transition Étiqueté* qui est son interprétation sémantique. Ce *Système de Transition Étiqueté* constitue en quelque sorte l'arbre d'exécution de la spécification sur une machine abstraite. Les plus connues de ces techniques sont les *automates d'états finis*, *les réseaux de Pétri*, et les techniques *d'ordonnancement temporel*.

Détaillons le modèle d'automate d'états finis, qui est le modèle le plus simple et le plus utilisé pour décrire les protocoles.

### 1.2.2 Les automates d'états finis

Un automate d'états finis peut être défini par un sextuple :

$$< Q, q_0, I, O, A, T >$$

où $Q$ est l'ensemble fini des états de l'automate, $q_0$ l'état initial de l'automate, $I$ l'ensemble fini des événements en entrée, $O$ l'ensemble fini des événements en sortie, $A$ la fonction de génération de l'événement de sortie de $Q \times I \rightarrow O$ et $T$ la fonction de transition d'états de $Q \times I \rightarrow O$. La figure 1.1 montre la représentation graphique d'un exemple d'automate d'états finis. Son interprétation est la suivante :

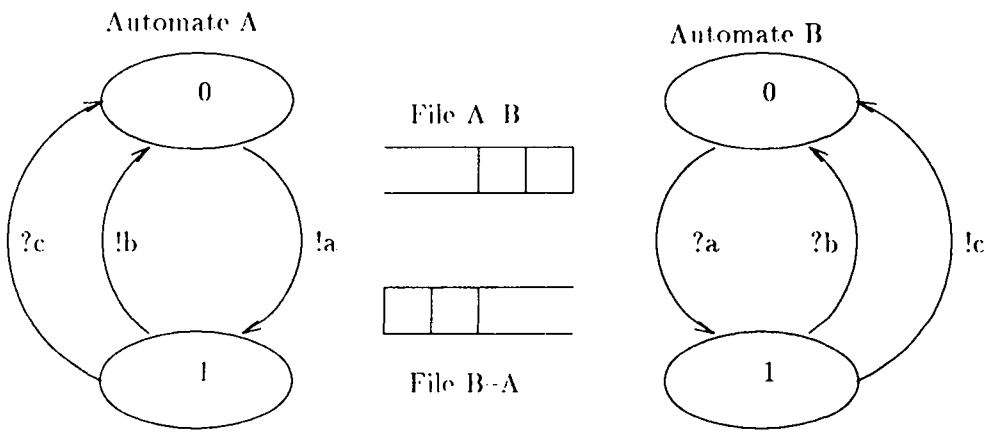


Figure 1.2: Etablissement d'une connexion entre deux processus

Si je suis dans l'état $q_1$ et que je reçois l'événement d'entrée $i$ alors je passe dans l'état $q_2$ en émettant l'événement de sortie $s$. On a dans ce cas: $Q = \{q_1, q_2\}$, $I = \{i\}$, O={ s }, $A(q_1, i) = s$, et $T(q_1, i) = q_2$.

Dans le domaine des protocoles, la sémantique donnée aux automates d'états finis est la suivante :

Une transition $t$ est tirable si je suis dans l'état d'entrée $q_1$ de cette transition et que je reçois l'événement associé $i$. Dans ce cas, on peut tirer cette transition (tirable) en passant dans l'état de sortie $T(q_1, i)$ de cette transition et en émettant l'événement associé $A(q_1, i)$. Lorsque plusieurs transitions sont tirables, il y a indéterminisme et la transition peut être choisie au hasard.

On peut maintenant modéliser un protocole à l'aide d'automates d'états finis : chaque entité communicante est définie par un automate, et les automates communiquent entre eux en s'échangeant des messages —les interactions pouvant s'effectuer par des files d'attente ou par rendez-vous.

Les automates peuvent être étendus par un jeu de variables locales (automate d'états finis étendus). On peut alors associer aux transitions des prédicats (fonction booléenne permettant de valider le tir d'une transition) et des actions qui mettent en jeu ces variables. Elles peuvent être aussi transportées dans les messages. Ces variables permettent par exemple de numéroter des trames d'information. Elles ont pour rôle de réduire notablement la complexité de l'automate. On peut aussi associer des priorités aux transitions qui peuvent remédier au non-déterminisme ainsi que des délais qui permettent de matérialiser la durée des échanges.

La figure 1.2 montre un exemple (dû à [13]) où deux processus A et B souhaitent entrer en communication. On suppose que seul A peut prendre l'initiative de la connexion en émettant le message $a$. Après un certain temps, A peut décider de fermer la connexion en émettant le message $b$. De même, B après avoir reçu le message $a$ peut aussi demander la déconnexion en émettant $c$.

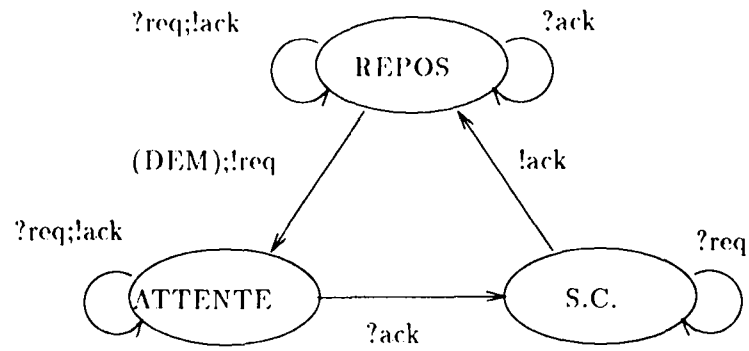


Figure 1.3: Protocole d'exclusion mutuelle entre deux processus

## 1.3 La validation de descriptions formelles

### 1.3.1 Graphe d'états

Le principe utilisé pour pouvoir valider une description formelle est de construire de manière exhaustive l'arbre de toutes les situations du système global par la méthode d'accessibilité des états. Une situation (ou *état global du système*) fait intervenir l'état de chaque automate, le contenu des files, et éventuellement les variables de chaque automate dans le cas d'automates d'états finis étendus.

$$situation = < etat, variables, files >$$

A l'élaboration de ce graphe, on peut vérifier des propriétés telles que :

- l'interblocage : états d'où les processus ne peuvent plus sortir,
- la réception non spécifiée (RNS) : événement se trouvant dans une file et non prévu par l'automate,
- le bouclage.

Si l'on reprenait l'exemple précédent (l'établissement d'une connexion entre deux processus de la figure 1.2) pour construire l'arbre des situations associées —en supposant que A et B communiquent par l'intermédiaire de canaux FIFO fiables non bornés— on obtiendrait alors un graphe de situation faisant ressortir une réception non spécifiée (RNS) d'où une mauvaise modélisation du problème (pour plus de détails, voir [13]).

Cependant, valider la consistance d'une description formelle n'est pas suffisant : on peut vérifier dans l'exemple de la figure 1.3 emprunté à [6] que la spécification d'un protocole d'exclusion mutuelle proposée ne produit ni interblocage, ni réception non spécifiée, ni bouclage. Pourtant, celle-ci n'est pas correcte, car on peut facilement constater que ce protocole n'assure pas l'exclusion mutuelle. Il faut donc un moyen d'exprimer les propriétés qui devront être respectées par une spécification, et de le vérifier. C'est l'intérêt des techniques de type logique.

### 1.3.2 Vérification de propriétés

On a donc besoin d'un langage de spécification pour énoncer les propriétés devant être satisfaites par un *Système de Transition Étiqueté*. Pour énoncer les propriétés d'un système, un langage déclaratif est naturel : ceci permet de spécifier le résultat et non pas la manière d'y parvenir. Un bon niveau d'abstraction est souhaitable puisque l'énoncé des propriétés devrait être le plus indépendant possible de la mise en œuvre. Ceci conduit naturellement vers l'utilisation de *logiques*.

En particulier, la logique temporelle introduite par Pnueli se présente comme une extension de la logique propositionnelle. Elle permet entre autres choses de spécifier des propriétés pour des protocoles de communication : non seulement les propriétés classiques d'absence de blocage, de boucles interdites, de terminaison mais également des propriétés de progrès qui se réfèrent au déroulement futur des événements.

Dans cette logique, le temps est vu comme une variable que l'on quantifie (existentiellement ou universellement). On trouve par exemple les opérateurs :

$\Diamond P$ pour dire qu'il existe un instant de temps futur pour lequel la propriété $P$ est vraie.

$\Box P$ pour dire que $P$ est continûment vraie dans le futur.

Avec cette logique, on peut aisément exprimer des propriétés telle que l'interblocage :

$$(b \Rightarrow \Box(b)) \wedge (P(init) \Rightarrow \Diamond(b)$$

*i.e.* le protocole ne progresse plus à partir de l'état $b$ et $b$ est accessible depuis l'état initial.

La vérification du système de transition étiqueté consiste à vérifier que tous les chemins du graphe satisfont ces propriétés. Ceci est résolu par des algorithmes de parcours du graphe spécifiques des opérateurs de la logique. La complexité en temps de la vérification peut être rendue linéaire en fonction du nombre d'états du graphe; elle est cependant généralement exponentielle en la taille de la formule.

## 1.4 La normalisation des langages formels de description de protocole

Très tôt dans le développement de l'OSI, il a été reconnu que les techniques de description formelle seraient utiles pour spécifier les protocoles. Le travail de l'ISO était motivé par le fait que les spécifications informelles comme celles données actuellement dans les normes ne peuvent éliminer toutes les ambiguïtés. Ces ambiguïtés peuvent en effet conduire à des mises en œuvre incompatibles et allant donc à l'encontre des buts de la normalisation. Ce paragraphe présente les travaux du groupe ISO/TC97/SC21/WG1 *ad hoc* FDT créé en 1980.

L'ISO est l'*International Standardisation Organisation*, le TC97 est le comité technique s'occupant de traitement des données, le SC21 est le sous-comité de TC97 s'occupant de l'interconnexion des systèmes ouverts, le WG1 est le groupe du SC21 travaillant sur les problèmes de modèle de référence de l'architecture, qui a mandaté le groupe *ad hoc* FDT (*Formal Description Technics*) composé de trois sous-groupes A, B et C.

Le sous-groupe A *Concept d'architectures* a pour mission de s'occuper des architectures pour les langages de description formelle développés par les sous-groupes B et C. Le sous-groupe B *Concept d'automate d'états finis étendus* a pour objectif de développer

une technique de description formelle fondée sur les automates d'états finis étendus: il a donné naissance au langage Estelle [9] étudié au paragraphe suivant. Enfin, le sous-groupe C *Concept d'ordonnancement temporel* a pour but de développer une technique de description formelle fondée sur l'ordonnancement temporel: il a donné naissance au langage LOTOS [8].

LOTOS (pour Language Of Temporal Ordering Specification) est un langage dont la syntaxe et la sémantique sont définies formellement. Il est fondé sur CCS (communication par *rendez-vous*) et les types abstraits algébriques comme définis dans le langage ACT-ONE.

Une spécification en LOTOS est une structure emboîtée de processus représentant les objectifs de la spécification. Ces processus peuvent être considérés comme des boîtes noires qui ont des portes (point d'interaction) pour communiquer entre eux. La communication se fait de manière synchrone par rendez-vous.

De son côté, le CCITT (Comité Consultatif International Téléphonique et Télégraphique) a défini les recommandations de son langage SDL [4] (*Specification and Description Language*, LDS en français) dans les avis Z101 à Z104, les premiers travaux de normalisation remontant à 1976. La recommandation définit deux formes de description sémantiquement équivalentes: une forme textuelle SDL-PR et une forme graphique SDL-GR (style d'organigramme).

LDS est fondé sur un formalisme d'automates d'états finis communicants, mais malgré l'introduction récente dans le langage des types abstraits algébriques, LDS reste un langage de la même génération que les organigrammes; aussi les grosses descriptions en LDS sont-elles peu structurées et difficiles à appréhender globalement. Il n'en demeure pas moins que LDS (dans sa version préliminaire sans types abstraits algébriques) est trés répandu dans le monde des télécommunications, et qu'il existe des outils pour générer directement du code depuis des diagrammes LDS [16].

## 1.5 Utilisation des techniques de description formelle

Nous avons vu l'importance des descriptions formelles de protocoles. La description en langage courant est nécessaire mais insuffisante car source d'ambiguïtés. Il est nécessaire de faire appel aux descriptions formelles pour:

- spécifier un protocole d'une façon claire et non ambiguë,
- avoir une base pour vérifier que la spécification est exacte (validation formelle),
- avoir une base pour l'implantation du protocole conformément à sa spécification.

Les techniques de description formelle ont tout d'abord été conçues pour permettre de spécifier les protocoles et les services des sept couches du modèle OSI, mais sont aussi utilisées pour décrire d'autres classes de protocoles et d'algorithmes distribués.

Des outils de validation de protocole ont d'ores et déjà fait la preuve de leur utilité. Ainsi par exemple VEDA (Validation et Évaluation d'Algorithmes Distribués, logiciel de simulation de protocoles décrits en Estelle développé au CNET de Lannion [11] et commercialisé par VERILOG à Toulouse) a permit de décrire précisément et de simuler des protocoles tels que le protocole session ISO ou le protocole X213 (qui intervient dans un

commutateur RNIS). De même, XESAR (outil de vérification de protocoles et d'algorithmes distribués fondé sur les travaux réalisés autour du projet CESAR —Conception Évaluation et Spécification des Applications Réparties— et élaboré en collaboration avec le CNET, le LGI et Cap Sogeti Innovation [17]) a permit la validation de protocoles comme le protocole de Stenning (généralisation du protocole du bit alterné avec $k$ messages en transit).

On pourrait encore mentionner EDT (ensemble d'outils Estelle Bull-INT), GEODE (éditeur —dirigé par la syntaxe— graphique (SDL-GR) ou textuel (SDL-PR), simulation avec génération de scénarios, et génération de code en langage *C* développé par la société VERILOG), OVIDE (description et analyse interactive un réseau de Pétri, développé par le LAAS et la société SYSECA), et bien d'autres développés au Etats-Unis, au Canada ou en Europe.

Ces techniques de description formelle sont toujours à l'étude dans les laboratoires de recherche et les résultats actuels sont prometteurs même s'il n'existe pas encore d'outil universel.

# Chapitre 2

# Le langage de description formelle Estelle

## 2.1 Introduction

Le langage de description formelle de protocole Estelle (Extended State Transition Language) est le résultat de recherches effectuées en parallèle aux Etats-Unis, au Canada et en France. La contribution française a été particulièrement importante grâce aux travaux sur PDIL (Protocol Development and Implementation Language) du projet RHIN de l'ADI (Agence De l'Informatique) et sur LC/1 du LAAS (Laboratoire d'Automatique et d'Analyse des Systèmes). Estelle est normalisé suite aux travaux du sous-groupe B du groupe FDT (Formal Description Technique) de l'ISO (ISO/TC97/SC21/WG1); la première proposition de norme remontant à 1985. Estelle est un langage qui permet de décrire des protocoles de communication et des algorithmes distribués, et dont la sémantique est définie formellement. Estelle repose sur deux choix principaux :

- le langage séquentiel PASCAL,
- le modèle d'automate d'états finis étendu.

## 2.2 Présentation générale du langage

Cette présentation s'appuie sur [3, 5, 7, 10] et sur la norme ISO [9].

### 2.2.1 Vue d'ensemble

La spécification en Estelle d'un système distribué est la description d'une collection hiérarchisée de sous-systèmes (ou *processus*) communiquant entre-eux. Ces sous-systèmes sont des sortes de *boîtes noires* à la SADT, munies d'interfaces typées spécifiées explicitements : leurs *points d'interactions*, ou *ports de communication*. Si l'on regarde à l'intérieur de ces boîtes noires, on peut trouver à nouveau une collection de boîtes noires (sous-systèmes *fils*) communiquant, ou bien un automate d'états finis étendu en décrivant le comportement.

Plutôt que de décrire directement ces sous-systèmes, on décrit en Estelle des modèles de sous-systèmes : les *modules*. Un sous-système (ou *processus*) est donc une instance (de

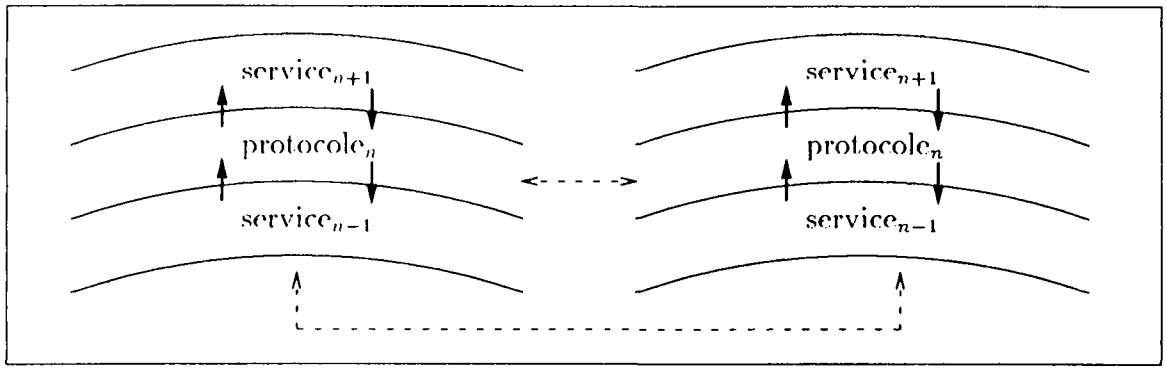


Figure 2.1 : Présentation classique d'un protocole OSI

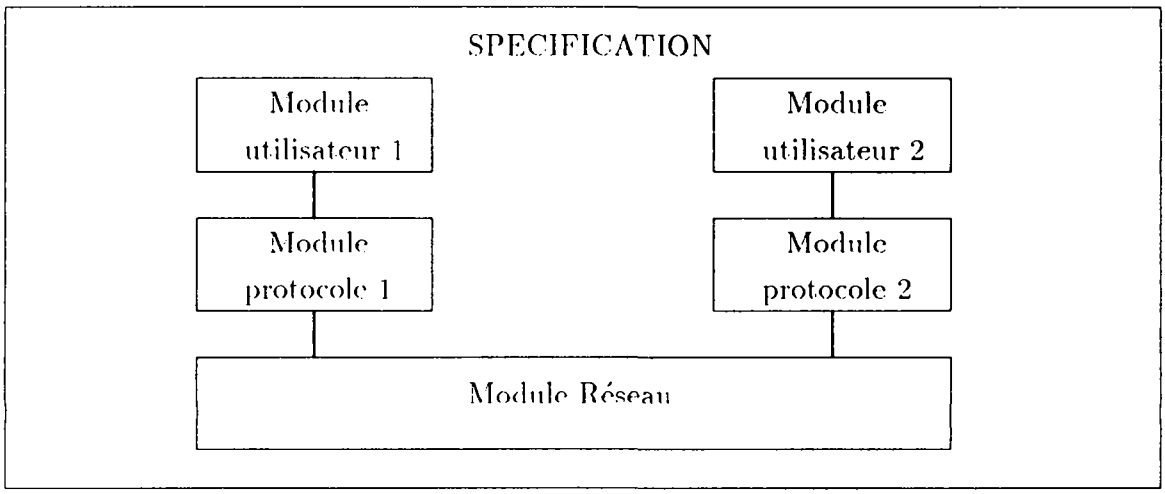


Figure 2.2 : Spécification décrivant un ensemble de modules

même qu'une variable Pascal est une instance d'un type) de *module*, qui peut-être crée, initialisée, interconnecté et détruite dynamiquement. Nous étudierons en détail au paragraphe 2.3 comment définir de façon abstraite un module, et au paragraphe 2.4 comment en spécifier le comportement.

La communication entre *processus* Estelle peut se faire à la fois par échange de messages (interactions) à travers des canaux (files FIFO non bornées) qui relient leur ports (points d'interaction), et par *export* de variables : certaines variables d'une instance de module fils (de la hiérarchie) peuvent être accessibles par son père (voir paragraphe 2.5).

La création dynamique des instances de modules à partir de leur définition abstraite et leur interconnexion (par canaux joignants leurs points d'interaction) sera présentée au paragraphe 2.6. Estelle permet de contrôler le type de parallélisme existant entre instances de modules : parallélisme synchrone ou asynchrone : nous le présentons au paragraphe 2.7.

### 2.2.2 Aspect général d'une spécification

L'aspect général d'une spécification Estelle (figure 2.2) reflète la présentation classique d'un protocole OSI (voir figure 2.1), où l'on trouve la notion de service rendu à la couche supérieure (ou *utilisateur*, *client*) en utilisant le service de la couche inférieure (*réseau*). Une description Estelle a la forme générale syntaxique suivante :

```
SPECIFICATION nom-spe
CONST          ...          -> Declaration de constantes
TYPE           ...          -> Declaration de types
CHANNEL        ...          -> Definition de canaux
PROCEDURE      ...          -> Definition des procedures et
                               fonctions de la specification
MODULE         ...          -> Definition de modules
MODVAR         ...          -> Declaration des variables
                               instances de modules
INITIALISATIONS...          -> Initialisations des instances de
                               modules, et connexions des canaux
```

A titre d'exemple, on trouvera en figure 2.3 la spécification Estelle correspondant au protocole de connexion-déconnexion de la figure 1.2.

## 2.3 Structure d'un module

Le langage Estelle permet de modéliser une machine d'états finis étendue (des états des transitions et des variables) tout en intégrant le langage structuré PASCAL. Un module est un modèle qu'on peut instancier (analogie PASCAL entre les types et les variables).

La spécification d'un module se fait en deux parties : d'une part la description de l'entête du module qui en définit les interfaces, et d'autre part la description d'un ou plusieurs corps (*body*) qui en donne le comportement, i.e. ses variables d'état et ses transitions.

### 2.3.1 Entête d'un module

Dans une entête de module on spécifie à la fois le type de parallélisme employé par ce module (voir paragraphe 2.7) et ses interfaces de communications : ses points d'interaction et variables exportées (voir paragraphe 2.5). Un module peut en outre être paramètré, ses paramètres devant être instanciés lors de son *initialisation*. Voici un exemple d'entête de module :

```
module M systemactivity (site : integer; b : boolean);
  ip p1 : T(r1) individual queue;
     p2 : U(r2) common queue;
     p3 : array [1..6] of V(r3) common queue;
  export X, Y : integer; Z : boolean;
end;
```

où $M$ est le nom du module, **systemactivity** son type de parallélisme, $p1$ un point d'interaction jouant le rôle $r1$ d'un canal de type $T$ (cf. 2.5) et muni d'une file d'attente individuelle, tandis que $p2$ et chacun des 6 ports du tableau $p3$ sont des points d'interaction ayant une file commune. $X, Y, Z$ sont les variables exportées par $M$ qui est en outre paramètré par *site* et $b$.

```
specification condecon;
default individual queue;

channel liaison(Nord,Sud);
by Nord : a;b;
by Sud  : c;

module type_A systemactivity;
ip pa : liaison (Nord);
end;
module type_B systemactivity;
ip pb : liaison (Sud);
end;

body comportement1 for type_A;
  state e0, e1;
  initialize to e0 begin end;
  trans from e0 to e1 begin output pa.a end; { dem. connexion}
  trans from e1 to e0 begin output pa.b end; { dem. deconnexion }
  trans when pa.c from e1 to e0 begin end;    { ind. deconnexion }
end;
body comportement2 for type_B;
  state e0, e1;
  initialize to e0 begin end;
  trans when pb.a from e0 to e1 begin end;    { ind. connexion}
  trans when pb.b from e1 to e0 begin end;    { ind. deconnexion }
  trans from e1 to e0 begin output pb.c end; { dem. deconnexion }
end;

modvar A : type_A; B : type_B;
initialize
  begin
    init A with comportement1;
    init B with comportement2;
    connect A.pa to B.pb;
  end;
end.
```

Figure 2.3 : Spécification Estelle du protocole de connexion-déconnexion

### 2.3.2 Corps de modules

Un module peut avoir plusieurs corps (*body*) ce qui revient conceptuellement à déclarer plusieurs comportements possibles pour ce module, l'association ne se faisant qu'à son initialisation. Les modules ainsi déclarés ont la même visibilité externe : mêmes points d'interaction, mêmes variables exportées, etc.

Le comportement d'un module est donné par un automate (i.e. liste d'états et de transitions) étendu avec Pascal (variables et conditions de tirs de transitions) et communicant : des transitions peuvent être conditionnées par des réceptions de messages, et il est possible d'en envoyer.

Son état initial est donné dans une transition spéciale dite d'initialisation ; les actions associées aux transitions sont décrites par une instruction composée Pascal. Ces transitions sont *atomiques*.

La structure syntaxique générale pour les corps de modules est la suivante :

**const** <*déclaration de constantes*>

**type** <*déclaration de types*>

**channel** <*déclaration de canaux*>

**module** <*déclaration d'interfaces de sous-systèmes*>

**body** <*déclaration de comportements de sous-systèmes*>

**modvar** <*déclaration des instances de sous-systèmes*>

**var** <*déclaration des variables Pascal*>

**function** *et* **procedure** *comme en Pascal*

**state** <*déclaration des états de contrôle*>

**trans** <*liste des transitions d'état : couples* condition,action>

La déclaration du corps d'un module peut donc contenir d'autres modules : on les appelle *modules fils*. Dans une spécification, on peut avoir plusieurs modules organisés hiérarchiquement. La spécification globale conduit alors à une arborescence dont la racine sera la spécification elle-même.

## 2.4 Description des transitions

Une transition Estelle est composée d'une *garde* qui est la condition pour qu'elle soit tirable, et d'une *action* qui est effectuée de manière atomique quand la transition est tirée. Il existe deux types de transitions : les transitions de réception d'une interaction et les transitions temporisées - qui sont des sortes de transitions spontanées.

### 2.4.1 Gardes des transitions

#### Transitions de réception d'une interaction

La réception d'une interaction (i.e. d'un message) par un processus Estelle sur un point d'interaction n'est possible que si celle-ci est présente en tête de la file du point d'interaction considéré, et peut être conditionnée par l'état de contrôle —appelé aussi état majeur— du processus ainsi que par l'état de ses variables locales, et même par les paramètres du message. Cette transition à la forme syntaxique générale suivante, les clauses facultatives étant entre crochets et les mots-clés Estelle en gras :

**trans**
*[***name** *nom-trans]*
*[***any** *domaine* **do]**
*[***priority** *entier-naturel]*
*[***from** $Etat_{initial}$*]* *[***to** $Etat_{final}$*]*
**when** *nom-point-interaction.nom-interaction [(parametres-formels)]*
*[***provided** `tests sur les variables locales et paramètres du message`*]*
**begin**
*[*`Actions Pascal et émissions de messages`*]*
**end**;

**trans** est le mot clé introduisant une transition

**name** permet de repérer une transition à l'aide d'un nom symbolique

**any** est une abréviation permettant de décrire en une seule fois plusieurs transitions ne différant que par des indices de variations sur des domaines bornés (par exemple **any** `i:1..6 do...`)

**priority** est une clause permettant de définir un ordre de priorité sur les transitions d'un même module. Une transition est d'autant plus prioritaire que sa **priority** est petite. En l'absence de cette clause, la transition est la moins prioritaire.

**from** désigne l'état de départ de l'automate

**to** ne fait pas réellement partie de la garde mais plutôt de l'action associée au tir de la transition ; il permet de désigner l'état d'arrivée de l'automate

**when** permet d'énoncer une condition sur la disponibilité d'un message particulier en tête de la file associée au port spécifié.

**provided** permet d'énoncer une condition sur les variables locales du module, et/ou sur les paramètres transportés par le message en tête de la file associée au port spécifié.

**Transitions temporisées**

Ces transitions permettent de modéliser les temporisateurs qu'on trouve souvent dans les protocoles. Elles ont la forme suivante :

**trans**
*[***name** *nom-trans]*
*[***priority** *entier-naturel]*
*[***from** $Etat_{initial}$*]* *[***to** $Etat_{final}$*]*
**delay***(val-min , val-max)*
*[***provided** `tests sur les variables locales`*]*
**begin**
*[*`Actions Pascal et émissions de messages`*]*
**end**;

On appelle *transition spontanée* une transition temporisée sans clause **delay**. Les paramètres du **delay** sont des valeurs dynamiques indiquant le retard minimum et maximum du tir d'une transition : quand une transition gardée par un **delay** devient tirable, on doit attendre au moins *val-min* et au plus *val-max* unités de temps pour la tirer (si elle reste continûment tirable).

Bien que faisant intervenir des notions de temps, Estelle n'est pas un langage temps réel car son modèle sémantique ne dépend pas de la durée des actions des transitions.

### 2.4.2 Fonctionnement dynamique

Le choix de la transition tirée parmi l'ensemble des transitions tirables d'une instance de module est non déterministe. On définit l'ensemble des transitions tirables comme étant l'ensemble des transitions d'une instance de module dont :

- l'état de départ est l'état majeur courant de l'instance du module,
- la priorité est maximale,
- le prédicat est vrai,
- pour une transition de réception d'une interaction, si l'interaction attendue est en tête de file (nom-interaction en tête de la file associée au nom-port-interaction) ou pour une transition temporisée, si l'instant de temps est compatible (délai compris entre val-min et val-max).

### 2.4.3 Les actions

Une action associée au tir d'une transition est composée de deux parties : un changement d'état majeur introduit par la clause **to** décrit précédemment, et une instruction composée Pascal (*begin..end*).

Afin d'être le plus indépendant possible des implantations, le Pascal utilisé est le Pascal ISO niveau 0, sans entrées sorties ni *goto*, et sans effet de bord pour les fonctions Pascal présentes dans les gardes. Ce Pascal de base est étendu à l'aide de quelques instructions nouvelles :

```
{ ... }
trans from cvet, pcs_op to cvet
 when cvsup.dem_syn(inval,aucun_contexte)
  provided not aucun_contexte
    var invalide_par_cas : EnsembleDeRef;
   begin
     invalide(inval,invalide_par_cas);
     output cvsup.rep_syn(invalide_par_cas,false);
   end;

trans from cvet to pcs_op
 when cvsup.ace_syn
  begin libere_ref(false) end;

trans when cvsup.Decon to non_con
  begin end;
{ ... }
```

Figure 2.4 : Fragments de transitions Estelle

**output** *nom-point-interaction. nom-interaction [(paramètres formels)]*
qui permet d'envoyer un message (transportant éventuellement des paramètres) sur un point d'interaction. Exemple :

```
output port.message(255,true);
```

**all** *domaine* **do** *instruction*
Cette instruction permet de parcourir un domaine de valeur sans spécifier l'ordre dans lequel celui-ci doit être parcouru. Par exemple pour mettre à zéro les éléments d'un tableau :

```
all i : 1..3 do table[i] := 0;
```

`i` n'a pas besoin d'être déclarée comme variable locale du module.

**forone** *domaine* **suchthat** *prédicat* **do** *instruction1* **otherwise** *instruction2*

**exist** *domaine* **suchthat** *prédicat*
Permettent de réaliser une instruction de manière non déterministe pour l'une des variables d'un domaine vérifiant une certaine condition.

On pourra trouver en figure 2.4 un exemple de textes de transitions.

## 2.5 La communication entre les modules

### 2.5.1 Les canaux de transmission

Les instances de modules peuvent communiquer entre elles par des liens de communication qui sont des files bi-directionnelles non bornées et de discipline FIFO. Les files sont

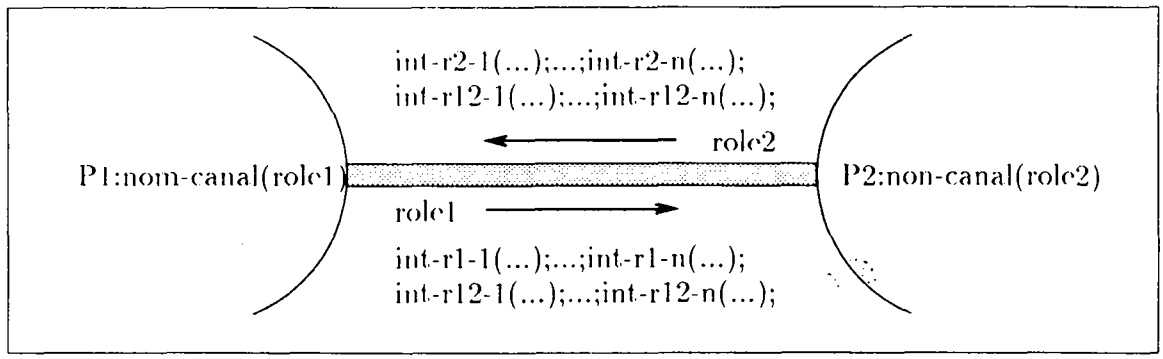


Figure 2.5 : Canal reliant deux points d'interaction P1 et P2

associées à un ou plusieurs points d'interaction ce qui définit deux possibilités :

- une file peut être associée à un seul point d'interaction (INDIVIDUAL QUEUE),
- une file peut être associée à plusieurs points d'interaction (COMMON QUEUE)[1].

Mais de même qu'on ne définit que des modèles de processus (les modules), un canal (CHANNEL) n'est en Estelle que le modèle d'un lien de communication.

### Définition des canaux

La spécification d'un canal contient une énumération de toutes les interactions (i.e. messages typés pouvant transporter des variables Pascal) pouvant transiter dans chaque sens du canal. Un canal est donc orienté, et ses deux extrémités jouent des *rôles* différents. La syntaxe de la déclaration du canal illustré en figure 2.5 est la suivante :

*CHANNEL nom-canal (role1,role2);*
*BY role1 : int-r1-1(...);... ;int-r1-n(...) ;*
*BY role2 : int-r2-1(...);... ;int-r2-n(...) ;*
*BY role1, role2 : int-r12-1(...);... ;int-r12-n(...) ;*

### Points d'interaction

Au niveau de la description d'un module, on définit la liste des points d'interaction auquel on associe le canal correspondant. On a une sorte de typage des points d'interaction par les canaux, qui définissent ce qui peut transiter en sortie, en entrée et dans les deux sens par les ports se référant à ce canal. Exemple correspondant au module de gauche représenté en figure 2.5 :

`IP p1 : nom-canal(role2);`

Cela signifie que le point d'interaction p1 est de type `nom-canal` dans lequel il joue le role `role2`. Le p1 pourra envoyer les interactions associées à `role2`, i.e. *int-r2-1...int-r2-n*

[1] Une politique par défaut est à préciser en début de spécification, par exemple `default individual queue`

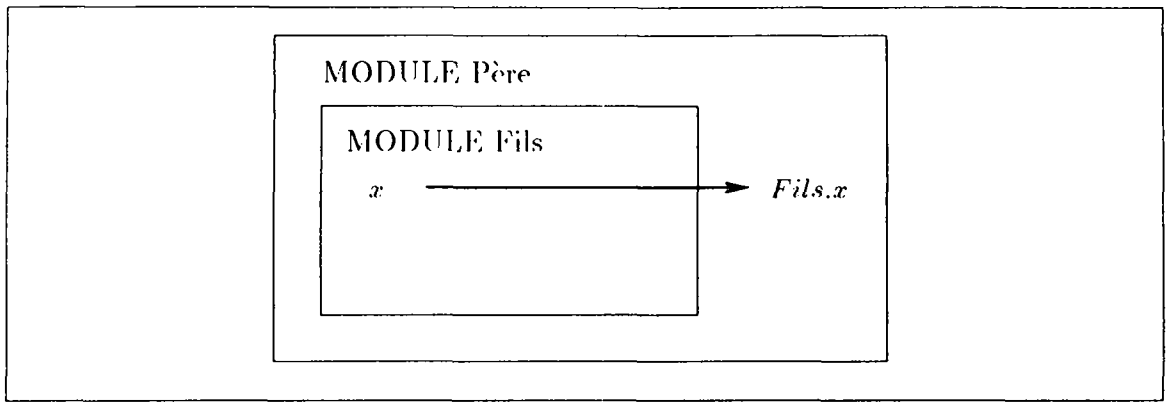


Figure 2.6 : Exemple d'exportation d'une variable $x$

et *int-r12-1...int-r12-n* et pourra recevoir les interactions associées à **role1**, i.e. *int-r2-1...int-r2-n* et *int-r12-1...int-r12-n*.

### 2.5.2 Les variables exportées

Au niveau de la description d'un module, on peut exporter des variables de la façon suivante : `EXPORT liste de variables exportées.`

En effet un module a la possibilité d'exporter des variables vers son module père (voir figure 2.6) qui pourra ainsi lire et écrire dans ces variables (exportées). L'accès simultané à une telle variable est exclu par la règle de priorité père-fils, qui sera décrite au paragraphe 2.7.

## 2.6 Instances de modules

Après avoir défini un module (entité abstraite au même titre que les types dans les langages structurés), on va pouvoir en créer une ou plusieurs instances. Comme on l'a vu, une instance de module est une variable de type module déclarée par la clause `MODVAR`. Pour déclarer une instance `v` du module `M` :

```
MODVAR v : M;
```

On peut aussi déclarer des tableaux d'instances de modules, comme par exemple :

```
MODVAR grille : ARRAY [1..N,1..N] of M;
```

### 2.6.1 Initialisation d'un module

#### Création d'une instance d'un module

Il s'agit d'attribuer un comportement `b` à l'instance de module `v`, i.e. de démarrer le processus `v` :

```
INIT v WITH b (parametres-effectifs);
```

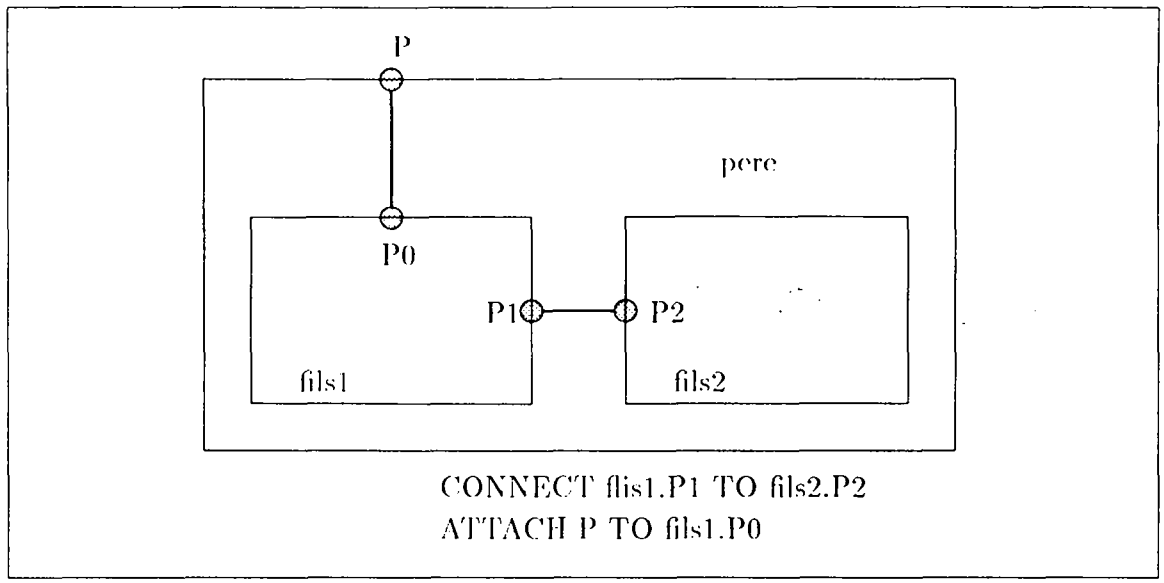


Figure 2.7 : Exemple de connexions entre instances de modules

#### Destruction d'une instance d'un module

`RELEASE v ;` où $v$ est la variable détruite

### 2.6.2 Connexion entre modules

Nous allons étudier maintenant comment connecter et déconnecter des instances de modules par l'intermédiaire de leurs points d'interaction.

#### Création de liens

On a deux modes possibles de connexion :

- si les modules à relier sont au même niveau :

```
CONNECT var-instance-module-fils.Point-interaction
     TO var-instance-module-fils.Point-interaction
```

- si les modules à relier sont à des niveaux différents :

```
ATTACH Point-interaction
    TO var-instance-module-fils.Point-interaction
```

La figure 2.7 montre un exemple de telles connexions entre des instances de modules `pere`, `fils1`, `fils2`.

### Destruction de liens

On peut de manière analogue détruire les liens créés:

- si les modules à relier sont au même niveau :

```
DISCONNECT var-instance-module-fils.Point-interaction
```

- si les modules à relier sont à des niveaux différents :

```
DETACH var-instance-module-fils.Point-interaction
```
ou
```
DETACH Point-interaction
```

## 2.7 Le parallélisme

Le comportement des modules les uns envers les autres dépend de leur place dans la hiérarchie et de leur attribut, qui peut être soit :

- systemprocess
- systemactivity
- process
- activity

Le déroulement de l'exécution de la hiérarchie des modules s'effectue en parallèle et respecte la règle de priorité parentale.

### 2.7.1 Priorité parentale

Cette priorité (Père Fils) signifie qu'une instance d'un module père est toujours plus prioritaire qu'une instance d'un module fils. Donc si on peut tirer à la fois des transitions d'un père et de ses fils, on tire en priorité celles du père.

### 2.7.2 Comportement des modules parallèles

On distingue trois sémantiques de parallélisme :

**Le parallélisme asynchrone** Ce type de parallélisme intervient entre modules étiquetés SYSTEMACTIVITY ; c'est le cas le plus général de parallélisme entre processeurs totalement indépendants, i.e. non-synchronisés : la mise en œuvre peut donc être distribuée.

Un schéma d'exécution possible serait de choisir de tirer, pour chaque instance de module, zéro ou une transition parmi ses transitions tirables —après avoir résolu les contraintes de priorité Père-Fils.

**Le parallélisme synchrone** Ce type de parallélisme intervient entre les sous-modules d'un module étiqueté SYSTEMPROCESS ; et permet de rendre compte d'un parallélisme maximal simulant la simultanéité de tous les événements. La mise en œuvre ne peut se faire qu'à l'aide d'un contrôle centralisé.

Un schéma d'exécution possible serait de choisir de tirer, pour chaque instance de module fils du module étiqueté SYSTEMPROCESS, exactement une transition parmi ses transitions tirables.

**Non déterminisme** Ce type de parallélisme intervient à l'intérieur d'un module étiqueté ACTIVITY ; et permet de rendre compte de l'entrelacement du tir des transitions. La mise œuvre peut se faire de manière séquentielle.

Un schéma d'exécution possible serait de choisir de tirer dans une instance de module fils du module étiqueté ACTIVITY exactement une transition parmi ses transitions tirables.

## 2.8 Conclusion

Estelle intègre donc le langage PASCAL en lui ajoutant des éléments qui en font un véritable langage pour l'expression de comportements parallèles. Estelle permet d'imbriquer des modules représentant un nombre quelconque de processus, permettant ainsi de bien structurer une spécification. Il s'adapte bien à la représentation des couches du modèle OSI. L'expression des structures de données en PASCAL permet de représenter des structures complexes : on peut par exemple décrire précisément les trames d'information et ainsi coder, décoder, concaténer et fragmenter les messages échangés.

Dès lors que l'on connaît le langage PASCAL et les automates d'états finis, l'apprentissage d'Estelle est relativement rapide. En effet il suffit principalement d'apprendre comment s'expriment la structuration des modules, les canaux et les transitions.

L'avenir d'Estelle semble prometteur mais dépend encore de l'existence d'un ensemble cohérent d'outils autour de ce langage, tel qu'un environnement logiciel pour la spécification, la validation et la mise en œuvre de protocoles.

# Chapitre 3

# Utilisation de ECHIDNA

## 3.1 Expérimenter des algorithmes distribués avec ECHIDNA

*ECHIDNA* est un environnement logiciel développé à l'IRISA pour aider à l'expérimentation des algorithmes distribués sur machines parallèles. L'expérimentation est une technique de validation intermédiaire entre la simulation aléatoire et l'implantation prototypique ayant pour but de simuler un protocole dans son environnement réel. *ECHIDNA* comprend les outils suivants :

- un compilateur du sous-ensemble dit *statique* du langage Estelle. Normalisé par l'ISO (IS 9074), Estelle est un langage fondé sur les automates d'état finis étendus avec le langage PASCAL
- des noyaux d'exécution distribués pour les machines Intel iPSC, FPS-T40, Telmat T-node, réseaux de Sun (3 et 4), autres machines Unix et même PC
- un ensemble d'outils logiciels pour observer de manière non intrusive le comportement de l'algorithme distribué qu'on "expérimente" (constructeur de temps global, traces, auto-enregistrement d'événements...)
- des facilités pour interfacer Estelle avec l'environnement de la machine sous-jacente
- un debogueur symbolique (au niveau source Estelle) multi-fenêtre interactif (un processus Estelle par fenêtre) disponible sous Suntools et X11/Motif.

Le compilateur *ECHIDNA* traduit la description formelle Estelle en un ensemble de structures de données (exprimées dans le langage de programmation *C*), qui décrivent totalement les constantes, types, modèles de canaux et de modules (associés à leurs corps) de la spécification. Cet ensemble de structures de données est ensuite compilé par le compilateur *C* local, puis lié avec le Noyau d'Exécution Distribué (NED) avant d'être chargé et exécuté sur le système cible.

Les autres fonctions du NED sont de fournir les primitives spécifiques d'Estelle et le *runtime* Pascal bâti à partir de celui du *C*. Il réalise aussi l'interface avec le système distribué sous-jacent, notamment pour le placement des tâches, la communication distante et les entrées-sorties. En pratique le NED, totalement écrit en *C*, est découpé en trois modules : l'interface avec le système, la machine virtuelle Estelle (Runtime), et l'exécuteur

(enchaînement des tâches). Autour du NED viennent se greffer des modules optionnels : simulation interactive, temps global, etc.

On pourra trouver une description plus détaillée de l'architecture du logiciel *ECHIDNA* dans [12, 15].

### 3.1.1 Notion de validation d'algorithme distribué

Devant la complexité des algorithmes distribués en tant qu'objets d'études, l'idée a très vite surgi d'utiliser des ordinateurs pour mieux les comprendre, les analyser et si possible pour tenter de les valider. Mais avant de faire appel à l'ordinateur, on doit absolument *modéliser* le phénomène qu'on désire étudier. Aussi, ne nous trompons pas : nous ne pouvons de toute façon qu'étudier des *modèles* d'algorithmes distribués. Le fait d'exprimer un algorithme distribué dans un formalisme proche de celui d'un langage de programmation nous permet seulement d'assurer que le modèle " colle " d'assez près à la réalité, c'est à dire qu'on va pouvoir implanter une version de l'algorithme distribué qui respecte le modèle.

De nombreuses équipes travaillent sur des outils de validation d'algorithmes distribués. Ces outils, fort différents dans leurs formes et leurs finalités, ont pourtant en commun d'accepter en entrée la description formelle d'une spécification d'algorithme distribué (ou protocole) et de fournir comme résultat un certain degré de confiance sur tel ou tel aspect ou propriété de l'algorithme distribué [1].

On distingue trois grandes classes de méthodes pour tenter de valider un algorithme distribué :

**La vérification formelle de propriétés** Son principe est d'explorer tous les états atteignables du protocole, par exemple en générant leur graphe comme dans Xesar [17], MEC [1] ou Meije [2] ; puis de vérifier des propriétés (spécifiées par exemple en logique temporelle) sur ce graphe. Cette méthode fournit un résultat sûr de validité mais présente de nombreuses limites d'applicabilité. Seuls des modèles simplifiés du protocole, correspondant à un niveau d'abstraction assez élevé peuvent être raisonnablement analysés par les méthodes actuelles. Ceci induit le problème (encore largement ouvert) de la préservation de la validité des propriétés au cours du raffinement du protocole : dans quelle mesure la description à un haut niveau d'abstraction reste-t-elle valable pour modéliser le protocole décrit plus finement?

**La simulation du protocole dans un environnement simulé** C'est une simulation généralement centralisée qui peut traiter des modèles d'algorithmes assez complets et permet de détecter efficacement des erreurs éventuelles sur un sous-ensemble des comportements possibles du protocole. La principale difficulté réside dans la nécessité de décrire et simuler l'environnement d'exécution de l'algorithme. Celui-ci est généralement très simplifié : il n'est pas réaliste de vouloir prendre en compte tous les paramètres d'un système d'exécution réel, comme par exemple l'influence précise de la taille des messages sur les temps de transmission, ou la durée des actions (non calculables sans exécution).

[1] En effet, si une erreur (*i.e.* un comportement non prévu par les spécifications ou bien en contradiction avec celles-ci) a été mise en évidence, on sait à coup sûr que l'algorithme est incorrect. Dans le cas contraire, on peut seulement lui accorder une certaine confiance, dépendante du niveau de détail auquel on est descendu, et de l'ensemble des états possibles du système qui ont été analysés.

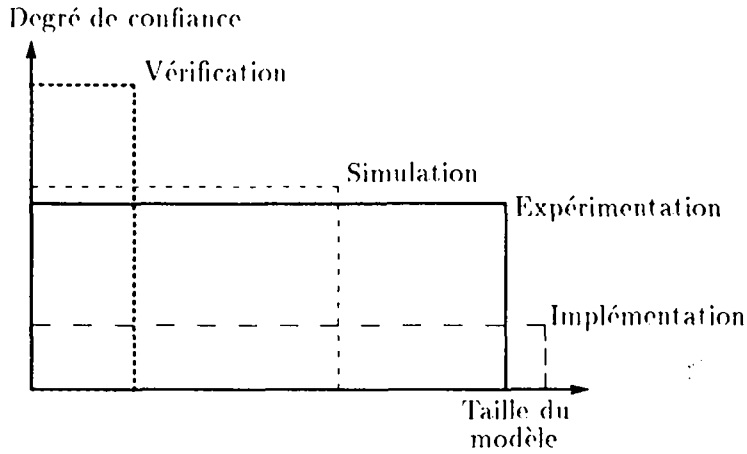


Figure 3.1 : La couverture des méthodes de validation

**L'observation et le test d'une implantation prototype du protocole** Bien que présentant l'avantage d'utiliser un environnement d'exécution réel, cette approche doit surmonter deux difficultés : l'absence d'outils pour observer un système réparti dans son ensemble, et le fait que le comportement du prototype peut dépendre étroitement de la mise en œuvre (par exemple, la résolution du non-déterminisme). Aussi il est difficile de tirer des conclusions générales quant au comportement de l'algorithme distribué réel.

Ces différentes approches, on le voit, sont complémentaires et doivent être utilisées de concert par le validateur. Elles ne recouvrent cependant pas tous les aspects de la validation des algorithmes distribués : il existe un véritable fossé entre notamment la simulation centralisée où l'on peut tout contrôler et observer et l'implantation où l'on est confronté à une réalité sur laquelle on n'a pas toujours de prise. Pour tenter de combler ce fossé, nous proposons de définir et de justifier une technique intermédiaire que nous appellerons *expérimentation*.

### 3.1.2 L'expérimentation, un aspect de la validation des algorithmes

L'expérimentation a pour but de contribuer à valider des algorithmes dans un environnement réel (système réparti), en explorant le plus grand nombre de leurs comportements possibles. Certains paramètres n'ont plus besoin d'être simulés : ils sont fournis par la machine parallèle elle-même. Si cette machine est suffisamment générale et son environnement bien maîtrisé, on pourra espérer transposer certains aspects du comportement de l'algorithme distribué expérimenté à n'importe quel type de machine, en quelque sorte *modulo* la machine d'expérimentation.

Stanislas Ulam explique dans son autobiographie [18] ce qu'il entend par expérimentation numérique :

> *L'objectif escompté est d'obtenir des lumières sur des questions de mathématiques pures. En produisant des exemples et en observant les propriétés de cer-*

*tains objets mathématiques, on peut espérer obtenir des éléments de réponse quant au comportement des lois générales.*

Par delà l'aspect validation d'algorithmes distribués, nous attendons de l'expérimentation qu'elle nous conduise à la découverte de comportements inattendus, et éventuellement à la remise en cause d'hypothèses jusque là admises.

Il faudra donc que l'expérimentation d'algorithmes distribués sur machines parallèles puisse nous donner des indications sur les performances de ces algorithmes par rapport à des critères d'efficacité ("speed-up" ou mesures plus évoluées de parallélisme, encombrement mémoire) ou de qualité (taux d'utilisation des ressources, taux d'erreurs de transmission récupérées par le protocole, charge du réseau, répartition de la charge entre les processeurs, conflits d'accès, vérification de propriétés). Une application possible serait de pouvoir tirer des informations sur les algorithmes distribués indépendamment de la machine cible, ou au contraire des informations sur l'ensemble machine+système d'exploitation indépendamment des algorithmes. Ceci doit permettre notamment de comparer les uns par rapport aux autres des choix de réalisations (choix structurels ou plus simplement dimensionnement de constantes) pour des classes d'algorithmes distribués rendant le même type de service, sur d'autres critères que leur complexité théorique en temps ou en nombre de messages.

Un avantage de l'expérimentation sur machine parallèle est que par construction on tire parti de la puissance de calcul de ce type de machine, ce qui permet d'envisager l'analyse et la validation d'algorithmes comportant plusieurs dizaines de milliers de processus.

L'inconvénient principal en découle : il faut mettre en œuvre des techniques spéciales pour pouvoir effectuer des prises de mesures et de l'observation distribuées. De plus, il faut impérativement prévoir un traitement automatique pour dépouiller la masse de résultats récoltés, grâce par exemple à des outils automatiques de vérifications de propriétés ou de visualisation graphique.

### 3.1.3 Une méthode d'utilisation pour l'expérimentation

Pour conclure cette présentation de l'expérimentation d'algorithmes distribués sur machines parallèles, donnons un exemple de ce que pourrait être une session typique de conception et de réalisation d'un algorithme distribué à l'aide d'*ECHIDNA*.

Tout d'abord il est sans doute souhaitable de n'exprimer en Estelle que la partie contrôle de l'algorithme distribué, et le reste dans le langage de son choix (remarquons que dans le cas d'un protocole de type OSI, il n'y a pratiquement que du contrôle...).

On peut alors commencer à travailler sur la preuve formelle (outil de type Xésar) d'un modèle de haut niveau et assez abstrait de l'algorithme, puis sur des preuves formelles partielles de sous-ensembles très précis du protocole. Ceci peut être facilité par quelques aller et retour vers des outils de type éditeurs graphiques ou syntaxiques et de simulation (type Véda), pour la mise au point (syntaxe, parties séquentielles...) et aussi pour avoir une première vision des comportements possibles du protocole.

Dans un deuxième temps, on pourra effectuer son expérimentation (à l'aide d'*ECHIDNA* par exemple) sur différentes machines parallèles (avec éventuellement réexécution) pour faire des mesures et des comparaisons entre différentes versions de l'algorithme. Après divers réglages, ajustement de paramètres et autres choix d'options, on pourra lancer de très grosses simulations (pour simuler l'ensemble d'un réseau par exemple), et finalement envisager une implantation automatique si cela est possible.

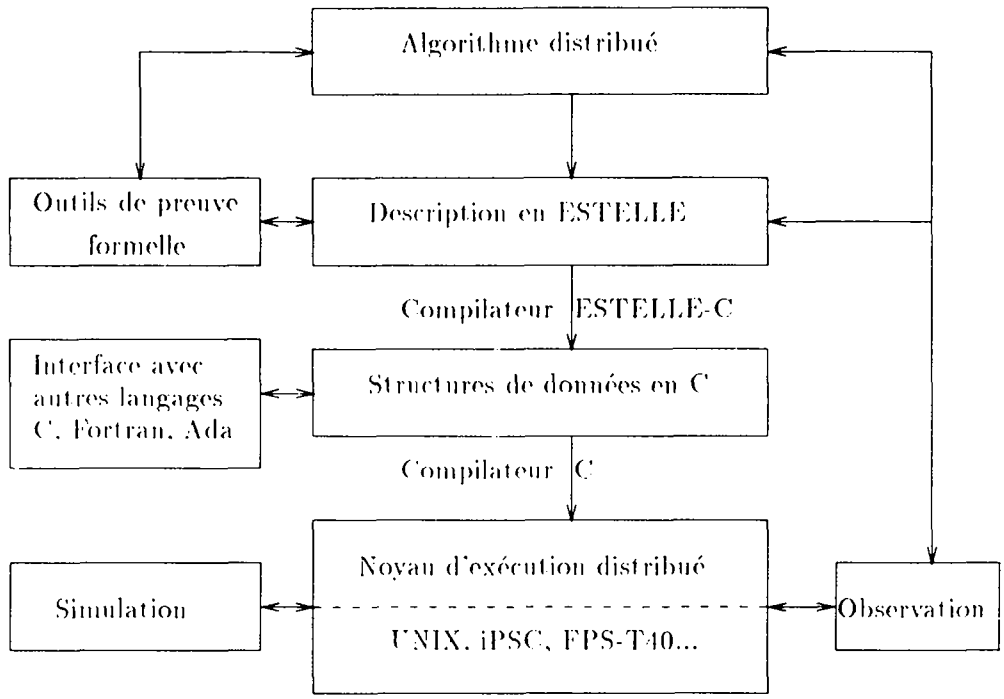


Figure 3.2: Expérimentation d'algorithmes distribués sur machines parallèles

Pour finir, précisons que toute utilisation d'*ECHIDNA* non conforme à la philosophie initiale des auteurs est bien sûr vivement encouragée!

## 3.2 Le modèle Estelle de *ECHIDNA*

### 3.2.1 Restrictions par rapport à la norme

Le modèle d'Estelle adopté par *ECHIDNA* est un sous-ensemble de la norme ISO. Dans *ECHIDNA*, une spécification Estelle est la description du comportement d'un système fermé, c'est à dire sans interaction avec l'extérieur. Un système est soit un ensemble de sous-systèmes s'exécutant en parallèle asynchrone, soit un ensemble de tâches s'exécutant en parallèle (à ce niveau, le parallélisme peut être synchrone ou asynchrone).

Une description Estelle est un arbre dont les nœuds sont des tâches et la racine la spécification englobante (voir figure 3.3 ). Les fils d'une tâche sont les instances des modules (variables de type *module*). La structure de cet arbre est dynamique.

Dans *ECHIDNA* cependant, nous nous sommes limités au sous-ensemble d'Estelle, qui permet toujours la création dynamique de l'arbre, mais pas sa modification. Cette limitation, aussi mise en œuvre dans Véda [11] ne semble pas trop restrictive au niveau d'abstraction où nous nous situons, car la reconfiguration dynamique n'est le plus souvent utilisée que comme facilité d'écriture et pour optimiser la mémoire d'une implantation. De plus, plusieurs concepts d'Estelle deviennent alors caducs : le partage de variables entre fils d'un module n'est plus possible, ce qui rend inutile une règle importante d'Estelle, la

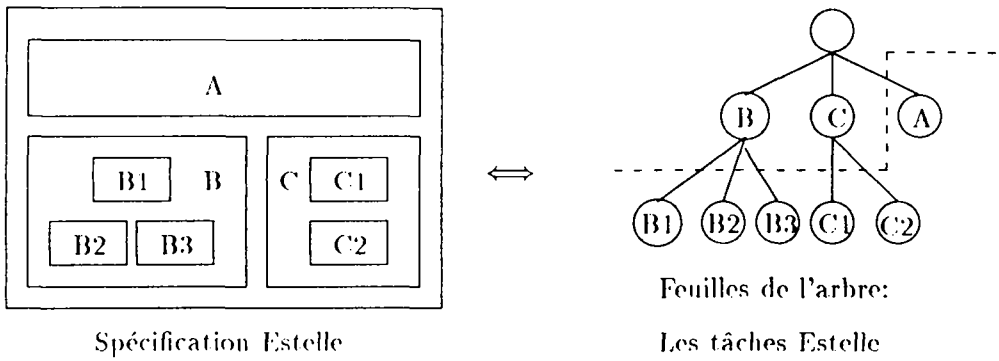


Figure 3.3 : Equivalence boîtes imbriquées à la SADT et structure arborescente

priorité père-fils. En outre, la distinction entre processus et activités —qui précise la nature du parallélisme— est traité de manière différente, le choix ne se faisant qu'à l'exécution (cf infra).

La dernière restriction majeure portant sur le langage concerne la clause *delay*. Le temps ne s'écoulant pas partout de la même façon dans un système réparti, nous ne pouvons pas implanter le *delay* Estelle tel quel. Mais, pour pouvoir observer des exécutions nous avons bâti un service de temps global permettant d'estampiller les événements observables (option de compilation *-G*). Nous envisageons que ce temps global puisse être utilisé pour implanter le délai Estelle avec une sémantique de temps global si nécessaire.

Il y a aussi quelques restrictions d'ordre syntaxique : les instances de modules doivent être initialisées avant d'être connectées, les *common queues* ne sont pas autorisées et l'emboîtement des procédures et des transitions n'est pas permis.

### 3.2.2 Estelle sur machines parallèles

#### Objectif

L'objectif qui nous a guidé a été de faire "coller" la description Estelle à la machine telle qu'elle est en réalité, plutôt que de reconstruire autour de n'importe quelle machine un environnement pesant destiné à implanter une sémantique particulière du langage Estelle. C'est pourquoi nous définirons seulement notre modèle de *station d'expérimentation* comme un ensemble de sites (au sens de *machines de Turing*) communicant exclusivement par échanges de messages au travers d'un réseau de communications point à point. Cette machine virtuelle possède en outre les propriétés suivantes :

- les différents sites évoluent *a priori* à des vitesses différentes.
- il n'y a pas de perte de messages : le temps de transfert des messages est fini mais non prévisible.
- les canaux de communication sont FIFO : les messages ne se dé-séquencent pas sur un canal.

Bien sûr, un problème risque de surgir si l'on veut mettre en œuvre cette abstraction de machine. Par exemple, une machine réelle ayant une mémoire finie, la machine de Turing (à mémoire infinie, bien pratique pour permettre des canaux de communication de taille non bornée) n'est pas réalisable dans le cas général. Mais un programme qui aurait besoin d'une mémoire infinie (ou de files d'attentes infinies) pour fonctionner serait d'une bien piètre utilité. En pratique, *ECHIDNA* tente mettre en œuvre dans la mesure du possible le modèle de *station d'expérimentation*, et de détecter la sortie du modèle, c'est à dire le fait que la machine réelle (équipée de son système) ne se comporte plus comme l'abstraction - par exemple lorsqu'il n'y a plus de mémoire disponible à cause d'une file de messages devenant infinie (ou d'une autre raison), ou lors d'une perte ou altération de message (très rare), ou lorsque la température monte trop ou que le plafond s'écroule... En cas de sortie du modèle, la seule chose qu'on pourra dire est que l'expérience n'est pas concluante, car l'outil d'expérimentation est trop limité.

Comme *ECHIDNA* doit générer du code pour un système distribué, une tâche terminale (feuille) n'est réellement démarrée sur un site donné que si elle doit y être exécutée. Les tâches non terminales sont passives : leur automate est réduit à une simple initialisation. Finalement, il ne reste sur chaque site que l'ensemble des tâches terminales (les feuilles de l'arbre) créées et placées dynamiquement sur ce site. L'exécution du programme Estelle va donc consister à partir de là à exécuter en parallèle les tâches sur chaque site.

### Placement des processus Estelle

Le placement des processus sur les sites physiques est explicitement défini par le programmeur Estelle. Si le premier paramètre d'un module est de type entier, *ECHIDNA* l'interprète comme le numéro sur lequel l'instance de ce module doit être exécuté ; l'association étant effective lors de la création du processus (i.e. initialisation). Les processus sont affectés par défaut sur le site de numéro 0 ; chaque site étant supposé avoir une identité unique (un entier positif), et il existe un site 0. Une machine centralisée est vue comme le site 0 d'un réseau à un seul site.

Dans le cas où le programmeur utilise plus de sites qu'il n'y en a d'effectivement disponibles lors du chargement du programme, le système effectue une sorte de *replicment* du graphe des sites *logiques* du programmeur sur le graphe des sites *effectifs* de la machine en prenant le numéro de site logique *modulo* le nombre total de sites *effectifs*[2].

Le nombre de tâches Estelle exécutées sur la machine parallèle pouvant être *a priori* différent du nombre de sites de cette machine, nous devons distinguer deux niveaux de parallélisme : le parallélisme inter-nœuds qui est le vrai parallélisme de la machine et le parallélisme intra-nœuds ou *quasi-parallélisme* : ce parallélisme est simulé sur chaque site par un *enchaîneur de tâches (scheduler)* dans le cas où plusieurs tâches Estelle sont démarrées sur un même site physique.

### Sémantique d'exécution

Un des intérêts de l'expérimentation sur machines parallèles étant d'observer le comportement d'algorithmes distribués dans un environnement réel, nous avons choisi pour

[2]Ceci permet d'exécuter n'importe quel programme Estelle *ECHIDNA* sur une topologie complètement maillée, et donc sur un système centralisé

sémantique d'exécution d'un programme Estelle la sémantique sous-jacente du parallélisme de la machine parallèle considérée.

Pour ce qui est du parallélisme simulé sur un site, nous avons choisi une sémantique dite des *lots parallèles* telle que la définit R. Groz en annexe de sa thèse [6]. Comme le montre R. Groz, cette sémantique permet d'explorer plus de comportements qu'une sémantique d'entrelacement (possibilité de croisements des messages), et en outre elle est plus " efficace " en termes de nombre de transitions tirées par unité de temps, car on n'a plus besoin de réévaluer les gardes des transitions avant d'avoir exécuté toutes les actions d'un lot[3].

Nous avons cependant laissé à l'utilisateur la possibilité de *paramètrer* cette sémantique d'exécution grâce à des options de la ligne de commande (à la UNIX, voir le manuel d'utilisation en annexe A). Il peut ainsi à volonté choisir de simuler de l'asynchronisme entre plusieurs processus d'un même site (option d'exécution *-a*), et/ou un choix déterministe de la transition à tirer parmi les transitions tirables d'un processus (option d'exécution *-D*).

## 3.3 Compiler et exécuter un programme Estelle avec *ECHIDNA*

La commande appelant le compilateur *ECHIDNA* est **estl**, sa syntaxe est proche de celle du compilateur *C* de Unix. Si l'on entre la commande **estl** sans paramètres, on obtient la notice d'utilisation décrite en annexe A.

### 3.3.1 Compilation

Après avoir écrit une spécification Estelle à l'aide d'un éditeur de texte, on peut obtenir un programme exécutable à l'aide de la commande **estl** dont voici la syntaxe :

```
estl [-bcCtSGHIX] [-s sys] [-m file] [-l] [-o objFile] [EstelleFile]
```

Le programme **estl** appelle le compilateur *ECHIDNA* pour traduire une description formelle Estelle en un ensemble de structures de données exprimées dans le langage *C*, puis les compile à l'aide du compilateur *C* local, et enfin les lie avec un Noyau d'Exécution Distribué (NED) pour produire un exécutable.

Si une erreur est découverte dans le fichier source Estelle, **estl** affiche un message d'erreur et essaye d'indiquer les constructions syntaxiques qui étaient attendues. Il suffit alors de taper < *RETOUR* > pour que **estl** appelle votre éditeur de texte favori (son nom doit être mis dans la variable d'environnement EDITOR, sinon c'est *vi* qui est appelé) et place le curseur sur la ligne erronée. **estl** reprendra la compilation lorsque vous quitterez l'éditeur. Les options de compilation sont les suivantes :

**-b** Mode non interactif : **estl** ne propose pas d'appeler l'éditeur lorsqu'une erreur est rencontrée ; mais affiche simplement un message d'erreur et termine.

**-c** Réalise seulement la traduction Estelle vers *C*

[3] En l'absence de clause de priorité sur les transitions, remarquons que les sémantiques d'entrelacement et des lots parallèles sont équivalentes.

**-C** Effectue la compilation *C* et l'édition avec le NED. Cette option n'est utile que si précédemment on s'est arrêté après la traduction Estelle vers *C* (option -c).

**-t** Empêche le compilateur de produire le code vérifiant les débordements de tableaux Pascal. Attention, l'utilisation de cette option peut empêcher le système de détecter ce type d'erreur, ce qui peut provoquer des "plantages" catastrophiques.

**-S** Demande l'inclusion de la table des symboles Estelle dans le code généré; ce qui est nécessaire pour certaines options d'exécution (cf infra).

**-G** Demande l'inclusion de la table des symboles *C* dans l'exécutable. Ceci n'est utile que si l'on désire utiliser un deboguer symbolique au niveau du langage *C*.

**-H** Demande la génération de code permettant d'établir un temps global avant l'exécution proprement dite de la spécification Estelle. Ceci revient à augmenter le temps de chargement du code exécutable (par défaut, le programme Estelle démarre alors simultanément sur chaque site au temps $t = 330$ secondes). Pour une description de l'algorithme utilisé, voir [14].

**-I** Le code généré permettra d'effectuer une simulation interactive multi-fenêtres de la spécification Estelle sous Suntools (voir paragraphe 3.4).

**-X** Le code généré permettra d'effectuer une simulation interactive multi-fenêtres de la spécification Estelle sous X11/Motif (voir paragraphe 3.4). *Cette option n'est pas disponible avant la version 3 de estl.*

**-s** Permet de désigner le système cible sur lequel devra s'exécuter la spécification Estelle. La compilation *ECHIDNA* doit bien sûr être faite depuis une machine supportant ce système. Les différents choix possibles sont actuellement (version 3.1):

**unix** (option par défaut) un système UNIX 4.2 ou 4.3 ou SYSTEM V.3

**netsun** un réseau de stations de travail Unix munies de TCP/IP (type Sun)

**ipsc2** un Hypercube Intel iPSC/2, à base de i80386 (fonctionne aussi pour l'iPSC/i860)

**fps** un Hypercube Floating Point Systems à base de Transputers

**helios** une machine Telmat T-node à base de Transputers munie du système *HELIOS*

**pc** Un PC compatible IBM (PC.XT.AT) sous MS-DOS. Le compilateur *C* utilisé par `estl` est MSC.

*Consulter le manuel en ligne (commande* `estl` *sans paramètres) pour avoir la liste complète des systèmes supportés)*

**-m** *fichier* Demande l'édition de lien avec un fichier objet (*fichier.o*) ou *C* (*fichier.c*). Cette option peut être répétée. Pour avoir plus d'information sur les possibilités d'interfaçage de Estelle avec *C*, voir le paragraphe 3.3.4.

**-l***librairie* Demande l'édition de lien avec une librairie. Cette option peut être répétée.

**-o** *NomExécutable* Permet de donner le nom *NomExécutable* au fichier exécutable produit par `estl`. Par défaut, celui-ci est *a_out*.

Par exemple pour compiler pour un réseau de Suns la spécification Estelle contenue dans le fichier `essai.stl`, lier avec le fichier *C* `util.c` et la librairie mathématique, et nommer l'exécutable `essai`, il faut entrer la commande :

```
estl essai.stl -o essai -s netsun -m util.c -lm
```

**ATTENTION :** Des programmes Estelle erronés peuvent ne pas être détectés par `estl`, et même franchir l'étape de la compilation *C*. Cette situation assez rare peut notamment apparaître quand on utilise des expressions complexes mettant en jeu des variables de types incompatibles.

### 3.3.2 Chargement et exécution

Il est possible de fournir des paramètres optionnels lors du chargement d'un exécutable compilé par `estl` :

-d*durée* Spécifie la durée d'exécution en secondes (un entier), i.e. une fois ce temps écoulé, l'exécution se termine sur chaque site. La valeur par défaut est 0, qui spécifie une durée infinie : dans ce cas il faut arrêter le programme en entrant le *kill-character*, usuellement $< CTRL - C >$.

-s*germe* Permet de spécifier un germe (un entier) pour le générateur aléatoire de *ECHIDNA*. Deux expérimentations *ECHIDNA* menées avec le même germe doivent produire les mêmes résultats s'il n'y a pas d'autre différence dans l'environnement.

**-b** Demande la bufférisation de toutes les traces de la spécification. Ces traces ne seront rapatriées qu'à la fin de l'exécution (si une durée à été spécifiée : option -d)

**-q** Inhibe l'impression des bannières et des avertissements (WARNING).

**-a** Demande la simulation d'une exécution asynchrone des processus Estelle à l'intérieur de chaque site (cf 3.2.2).

**-D** Sélectionne une sémantique d'exécution déterministe : la première transition tirable d'un processus Estelle est effectivement tirée, et les priorités sont ignorées (cf 3.2.2).

**-T** Demande la trace de l'émission et de la réception de chaque interaction échangée dans le système (l'option de compilation -S est nécessaire).

Pour charger et exécuter votre programme *pgm* après une compilation avec `estl` :

**Unix et MS-DOS** Entrer simplement :

```
pgm [options]
```

où `[options]` est la liste des options (voir ci-dessus) que vous avez choisies.

**Réseau de Sun** Il faut commencer par réserver (virtuellement) un ensemble de stations de travail (au plus 32) à l'aide de la commande :

```
getsun sun1 sun2 .. sunN
```

où **sun1 sun2 .. sunN** sont les noms symboliques de vos stations qui se voient affecter les numéros de sites 0,1,...,n-1[4].

Vous avez alors deux moyens différents pour lancer votre programme *pgm* sur le réseau de Sun :

- soit "à la main" depuis chacune des stations : dans ce cas, sur chaque station, il faut vous placer dans le répertoire d'où vous avez lancé le **getsun** puis entrer simplement :

  ```
  pgm [options]
  ```

  où **[options]** est la liste des options que vous avez choisies.

- soit exécuter la commande suivante sur une seule des stations (mais toujours depuis le répertoire d'où vous avez lancé le **getsun**) :

  ```
  loadsun "pgm [options]"
  ```

  où **[options]** est la liste des options que vous avez choisies. Cette commande lance automatiquement *pgm* sur chacune des stations sélectionnées, et toutes les traces sont redirigées vers votre terminal.

**Hypercube Intel iPSC/2 (et iPSC/860)** Il faut commencer par réserver un ensemble de processeurs à l'aide de la commande :

```
getcube [taille] [> nom_de_fichier]
```

où **taille** permet de spécifier la taille du cube réservé (par exemple **-t d3** réserve un cube de dimension 3 soit 8 nœuds) : voir le manuel de l'iPSC pour plus de précision. La redirection des traces du programme Estelle vers un fichier se fait aussi à ce niveau ; sinon elles apparaitront sur la sortie standard.

Vous pouvez alors lancer votre programme *pgm* sur l'hypercube en entrant la commande :

```
load pgm [options]
```

où **[options]** est la liste des options que vous avez choisies.

**T-node (Helios)** Il suffit, après avoir "booté" HELIOS et compilé avec les options adéquates, d'entrer la commande :

```
runec taille pgm [options]
```

où **taille** permet de spécifier la dimension du cube réservé (par exemple **runec 3** réserve un cube de dimension 3 soit 8 nœuds) et **[options]** est la liste des options que vous avez choisies.

[4] Cette commande crée dans votre répertoire courant un fichier *.net_topo* contenant le mapping (nom-symbolique,numéro de site).

### 3.3.3 Les prédéfinis de *ECHIDNA*

*ECHIDNA* permet l'utilisation des prédéfinis de la norme Estelle (les fonctions Pascal telles que *sin, cos, log, abs...*)[5], et offre de plus les prédéfinis suivants :

**trace(format)** permet de sortir une trace (tamponnée ou non, voir paragraphe précédent) et possède la même syntaxe que le *writeln* Pascal.

**ntrace(format)** permet de sortir une trace (tamponnée ou non, voir paragraphe précédent) et possède la même syntaxe que le *write* Pascal, *i.e.* se comporte comme **trace** mais sans passer à la ligne.

**randomint(i1, i2 : integer) :integer** fonction retournant un entier choisi aléatoirement dans l'intervalle $[i1, i2]$, avec une distribution uniforme.

**randomreal(r1, r2 : real) :real** fonction retournant un réel choisi aléatoirement dans l'intervalle $[r1, r2]$, avec une distribution uniforme.

**time : real** Retourne un réel donnant le temps local (en millisecondes) écoulé depuis le lancement du programme sur le processeur physique en charge du processus Estelle appelant.

Pour des raisons de compatibilité avec d'autres compilateur Estelle, ces prédéfinis peuvent être déclarés *primitive*.

Lorsqu'on utilise l'option de compilation -H (voir ci-dessus), *ECHIDNA* calcule un temps global qui est accessible depuis la fonction :

```
gtime(var precision : real) : real
```

qui retourne un réel donnant le temps global (en millisecondes) écoulé depuis le lancement du programme Estelle, et met dans **precision** sa précision (en millisecondes). Cette fonction doit être déclarée *primitive* dans le programme Estelle.

### 3.3.4 Interface Estelle-*C*

*ECHIDNA* a été conçu pour pouvoir interfacer facilement un programme Estelle avec le système sous-jacent, et notamment avec toute librairie *C* d'un système de type Unix.

Par exemple, comme il n'y a pas d'Entrée/Sorties dans Estelle (donc pas dans *ECHIDNA*), on peut vouloir réaliser un module d'interface Estelle-*C* pour pouvoir ajouter des fonctions (dites *primitives*) dans une spécification Estelle. Ceci peut être réalisé très simplement, comme le montre l'exemple suivant, où l'on veut faire afficher une invite (*prompt*) pour lire un entier sur l'entrée standard :

```
(* A declarer dans le programme Estelle: *)
function lit_ent(prompt : char):integer;primitive;
```

Il faudra alors à la compilation lier avec un programme *C* (par expl. *io.c* cf plus bas) contenant la fonction correspondante (voir manuel estl) :

```
> estl foo.d -o foo -m io.c
```

[5]Sur certaines machines, il peut être nécessaire de lier avec la librairie mathématique *C* : *-lm*.

Grosso-modo, la traduction en *C* d'une déclaration Estelle *primitive* se fait en :

- mettant la premiere lettre en majuscule, le reste en minuscule, ce qui évite les collisions avec les identificateurs du *C*.
- les types simples Pascal *char*, *boolean*, *integer*, *real*, *packed array of char* sont traduits respectivement en *C* par *char*, *char*, *int*, *double*, *char **
- un paramètre est passé par valeur, sauf s'il est déclaré *var*, auquel cas il est passé par adresse (*&ident* en *C*).
- pour les cas particuliers, voir le code *C* généré (foo.c).

Donc, dans le cas de notre exemple, le code *C* generé par *estl* est :

```
extern int Lit_ent();
```

et l'appel de cette fonction en Estelle `i := lit_ent('>');` est traduit en

```
processus->i = Lit_ent('>');
```

Voici un fichier *io.c* utilisable avec cet exemple :

```
#include <stdio.h>
int Lit_ent(c:char)
{
        int temp;
        putchar(c);
        scanf("%d",&temp);
        return temp;
}
```

## 3.4 Simulation interactive avec ECHIDNA

### 3.4.1 La place de la simulation interactive parmi les méthodes de validation de protocoles

Alors que la vérification exhaustive a pour but de construire —de façon exhaustive— le graphe d'états du protocole sur lequel seront vérifiées des propriétés, la simulation parcourt ce graphe de façon aléatoire (absence de mémoire des situations passées), en essayant de vérifier des propriétés au fur et à mesure que les états sont rencontrés. Ce type de simulation est possible dans ECHIDNA avec une stratégie de simulation synchrone ou asynchrone, déterministe ou non-déterministe.

Alternativement, la simulation interactive a pour but d'observer le comportement d'un protocole en visualisant des éléments intéressants (variables, état, transitions, messages...) et de permettre à l'utilisateur d'agir sur le déroulement de la simulation (tirer des transitions). On parle aussi d'animation [6]. Nous gardons le terme de simulation interactive car cet outil permet de faire aussi de la simulation aléatoire avec différentes stratégies : synchrone, asynchrone, séquentielle. Ses fonctionnalités s'apparentent à celles d'un debogueur symbolique d'algorithmes distribués.

L'aspect interactif d'une simulation amène les questions suivantes : *Que représenter ?* et *Comment le représenter ?* Ces questions sont souvent évoquées lors de la conception de logiciels où les dialogues entre l'utilisateur et un système informatique ont un rôle prépondérant. Il faut éviter de tomber dans les excès suivants :

- représenter à l'écran le maximum d'éléments, ce qui a pour défaut de noyer l'utilisateur dans un flot d'informations lui masquant les données utiles du problème,
- représenter à l'écran trop peu d'information et obliger l'utilisateur à demander explicitement ce qu'il souhaite voir afficher.

Aussi, dans notre système un certain nombre de choix gérent automatiquement la gestion de l'affichage, mais l'utilisateur est prioritaire sur la modification de ces choix implicites.

### 3.4.2 Description générale

Cet outil de simulation interactive peut être utilisé soit avec le système de fenêtrage Suntools, soit avec Motif qui est une interface au dessus de X-windows (X11). Les fonctionnalités sont dans les deux cas essentiellement similaires ; mais peuvent différer sur des points de détail : nous le préciserons à chaque fois.

Après avoir compilé une description Estelle pour produire un exécutable de simulation interactive (voir paragraphe 3.3.1), l'exécution de ce programme fait apparaître trois types de fenêtres (voir figure 3.5) :

- la fenêtre simulation, qui regroupe principalement les paramètres de la simulation et les différents modes de simulation (interactive, aléatoire...) offerts à l'utilisateur ; cette fenêtre a pour bannière le nom de l'exécutable sous Motif, et 'SIMULATION' sous Suntools.
- les fenêtres des processus affichés, où figurent les informations (files, état courant, variables) du processus et les transitions tirables ; cette fenêtre a pour bannière le nom du processus (avec un indice pour les tableaux de processus) de la description Estelle.
- la fenêtre historique de la simulation qui enregistre sous une forme abrégée l'historique de la simulation ; cette fenêtre a pour bannière : `HISTORIQUE -> source.his`.

Lorsqu'on lance le programme de simulation interactive, le système reprend la simulation dans l'état où il était pour la description courante du protocole (si le fichier *source.int* existe sinon initialisation de la simulation).

ATTENTION : quand on change le source de la description Estelle il faut détruire ce fichier *source.int* avant de relancer la simulation interactive.

### 3.4.3 Le contrôle de la simulation

Le contrôle de la simulation s'effectue depuis la fenêtre principale de simulation. La signification de ses différents éléments est la suivante :

**Pas** Un pas correspond au tir d'une ou plusieurs transitions selon le mode de simulation. **Pas** indique le pas courant de la simulation.

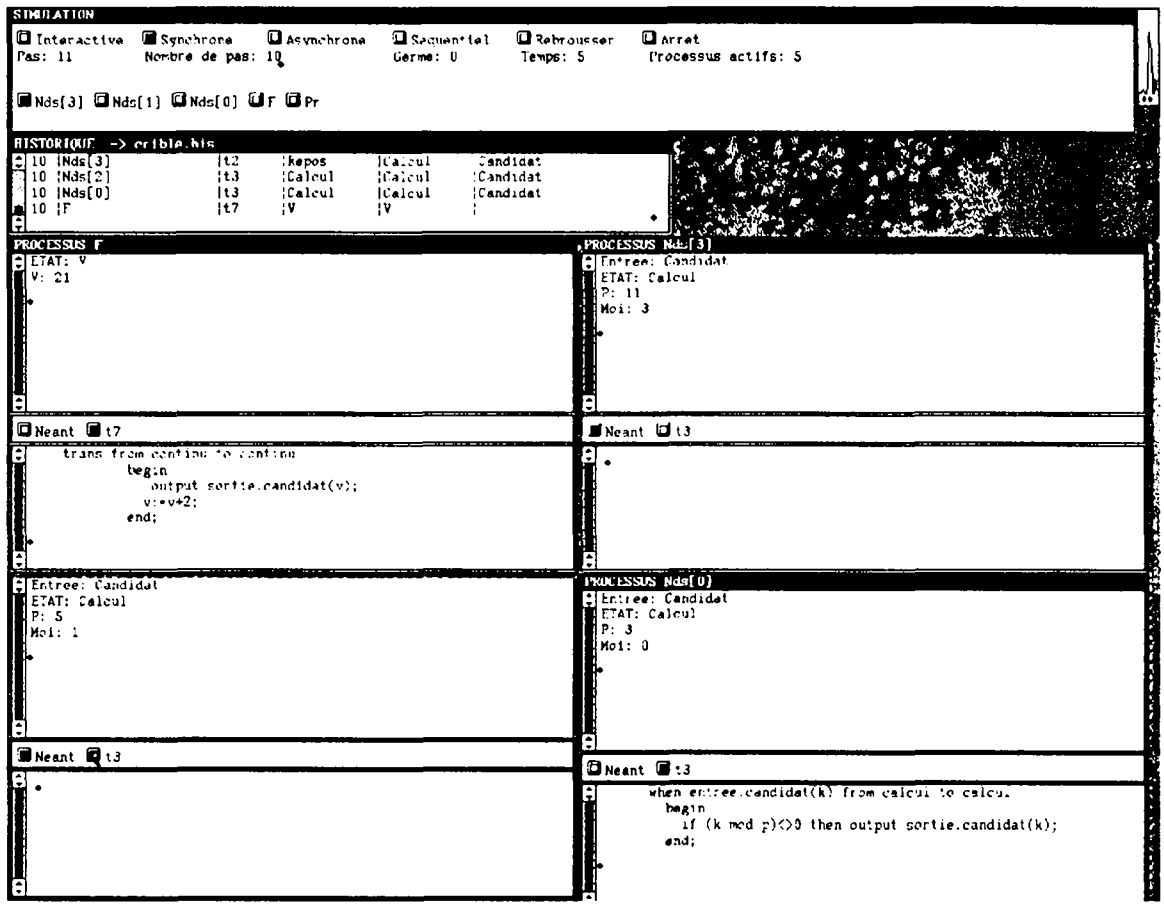


Figure 3.4 : Les fenêtres de la simulation interactive sous Suntools

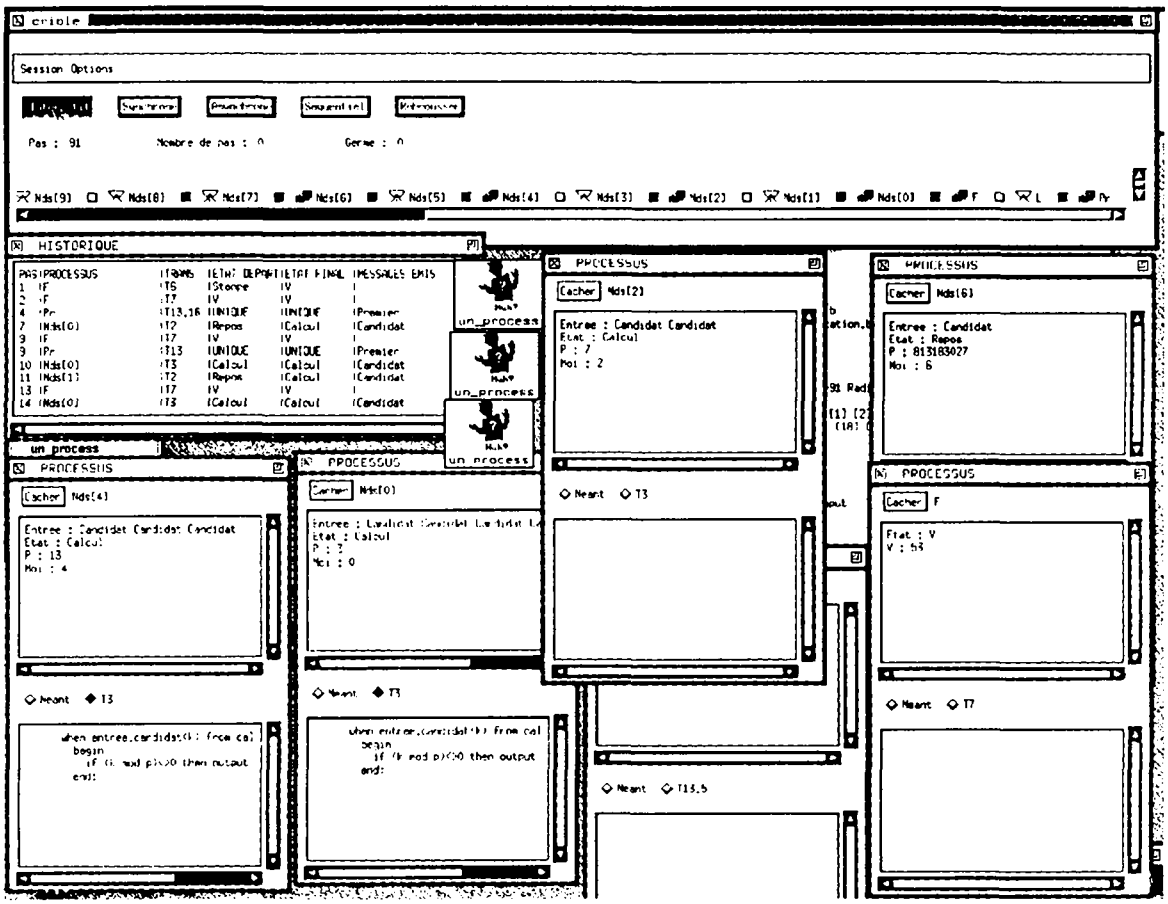


Figure 3.5 : Les fenêtres de la simulation interactive sous X11/Motif

**Nombre de pas** Cet élément intervient à la fois :

- au niveau de la simulation aléatoire (synchrone, asynchrone ou séquentielle) et a pour signification le nombre de pas que l'utilisateur souhaite jouer automatiquement,
- au niveau de la commande rebrousser, il a pour signification le nombre de pas que l'utilisateur souhaite voir revenir en arrière.

**Germe** Le germe de la simulation permet de rejouer exactement les mêmes situations. C'est un nombre entier que l'utilisateur met à jour (par défaut 0) et qui intervient en simulation aléatoire (synchrone, asynchrone ou séquentiel).

**Temps** (Suntools uniquement)
Le temps donne une indication sur le temps d'exécution machine de la simulation. Ce temps tient compte de l'évaluation des gardes des transitions des processus et du tir des transitions. 'Temps' indique le temps courant de la simulation. Lorsqu'on rebrousse ou repart d'une ancienne situation, le temps est mémorisé. Cette notion du temps est indicative et pourrait avoir une autre sémantique lors d'une nouvelle version d'*FCHIDNA*.

**Processus actifs** (Suntools uniquement)
Cet élément indique le nombre de processus actifs à un instant donné. Un processus est actif s'il offre au moins une transition tirable.

L'utilisateur a le choix entre différents modes de simulation : simulation interactive, simulation aléatoire synchrone, simulation aléatoire asynchrone, simulation aléatoire séquentielle, rebrousser, et mettre fin à la simulation en sauvegardant l'état courant.

Au cours d'une simulation, on peut passer à volonté d'un mode à l'autre. L'utilisateur sélectionne le mode de son choix en cliquant à l'aide de la souris sur le "bouton" correspondant dans la fenêtre de contrôle de la simulation.

**Simulation interactive** Dans la simulation interactive, l'utilisateur indique la ou les transitions qu'il souhaite tirer parmi l'ensemble des transitions tirables des processus proposés. Ce mode de simulation permet la mise au point pas à pas d'un protocole en examinant les cas intéressants et d'aider un étudiant à bien comprendre le fonctionnement d'un algorithme distribué. Au cours de la simulation, des informations apparaissent sur le comportement des processus. C'est simplement au moment de la sélection avec la souris du bouton **Simulation interactive** que les transitions sélectionnées au niveau des processus seront effectivement tirées (si l'utilisateur choisit un autre mode, les transitions sélectionnées ne seront pas forcément tirées).

**Simulation aléatoire** Dans ce mode automatique, l'utilisateur n'a aucune influence sur le déroulement de la simulation. A la fin de la simulation aléatoire, le système passe en mode interactif pour donner des informations à l'utilisateur. Ici la simulation est aléatoire et l'utilisateur indique :

- le germe du générateur de nombres ce qui lui permet par la suite de rejouer des situations identiques,
- et le nombre de pas à simuler en continu.

**Mode synchrone** Un pas de simulation aléatoire synchrone détermine pour chaque processus une transition choisie aléatoirement puis tire toutes ces transitions (une par processus).

**Mode asynchrone** Un pas de simulation aléatoire asynchrone détermine pour chaque processus une transition choisie aléatoirement puis tire un certain nombre de ces transitions (au plus une par processus).

**Mode séquentiel** Un pas de simulation aléatoire séquentielle détermine une transition choisie aléatoirement parmi les processus actifs puis tire cette transition.

**Rebrousser** Ce mode permet de revenir en arrière en cours de simulation. Cette option est très pratique pour élaborer de nouveau choix à partir d'un état connu ou simplement en cas d'erreur.

Lorsqu'on clique sur ce bouton, la simulation revient en arrière du nombre de pas indiqué dans le champ utilisateur *Nombre de pas*. Pour modifier ce champ, il suffit de cliquer dessus, et d'entrer la valeur voulue.

### 3.4.4 La gestion de l'affichage des processus sous Suntools

La gestion de l'affichage des processus est commandée soit au niveau de la fenêtre de la simulation, soit au niveau des processus (choix **Done** dans le menu du processus obtenu en cliquant avec le bouton droit dans le bandeau de la fenêtre du processus).

Tous les processus actifs ne sont pas visualisés à l'écran : on ne peut afficher simultanément que 4 processus parmi les processus actifs. Pour visualiser un processus actif non visible à un moment donné, il suffit de cliquer sur le bouton apparaissant à coté de son nom dans la fenêtre de contrôle de la simulation. Si on a plus de processus actifs qu'on ne peut en afficher dans cette fenêtre, un bouton *Processus suivants* est crée afin de pouvoir en parcourir la liste.

Lorsqu'on sélectionne un processus actif dans la fenêtre simulation :

- soit il existe une place libre pour afficher ce processus et alors il vient s'inscrire à cet endroit,
- sinon, il prend la place du premier processus affiché.

On peut enlever un processus de l'affichage en sélectionnant l'option *Done* du menu de la fenêtre du processus. Les transitions sélectionnées sont mémorisées. Cela permet de créer une place libre où l'on pourra afficher un autre processus.

Le système d'affichage des processus obéit à un certain nombre de règles :

- seul les processus actifs sont visualisés,
- les processus actifs sont placés par ordre d'apparition,
- un processus reste affiché à l'écran tant qu'il est actif,
- l'utilisateur est prioritaire vis à vis des règles précédentes,
- lorsque la simulation progresse d'un pas, le système respecte au mieux les règles précédentes.

### 3.4.5 La gestion de l'affichage des processus sous Motif

La gestion de l'affichage des processus est commandée soit au niveau de la fenêtre de la simulation, soit au niveau des processus. Par défaut tous les processus (qu'ils soient actifs ou pas) sont visualisés. Le bouton *Options* de la fenêtre de contrôle de la simulation permet cependant de cacher les processus inactifs (ceci prend effet à partir du pas suivant de la simulation).

Dans la fenêtre de contrôle de la simulation, on trouve à coté du nom de chaque processus soit:

- un bouton, noir si la fenêtre du processus correspondant est présente à l'écran, blanc sinon. Appuyer sur ce bouton permet de faire apparaître ou disparaître cette fenêtre.

- une icône représentant un sablier si le processus est inactif, un engrenage sinon.

On peut aussi faire disparaître la fenêtre d'un processus en cliquant sur son bouton *Cacher* (les transitions sélectionnées sont cependant mémorisées).

Par défaut les fenêtres apparaissent au centre de l'écran, légèrement superposées ; l'utilisateur peut cependant les disposer à son gré à l'aide des commandes habituelles de son gestionnaire de fenêtres X11.

### 3.4.6 Les fenêtres des processus

Détaillons maintenant le contenu des fenêtres des processus. Ces fenêtres sont découpées en trois sous-fenêtres contenant les informations d'un processus, la liste des transitions tirables et le texte de la transition couramment sélectionnée.

**les informations d'un processus :** dans cette sous-fenêtre déroulante figurent :

- la liste des ports du processus ayant des messages en attente. A coté du nom du port (avec éventuellement son indice dans le cas d'un tableau de ports) , on représente sa file d'attente en affichant les 2 premiers et le dernier message de la file, ainsi que leur nombre si tous ne sont pas affichés.
- l'état courant du processus : on affiche après le symbole ETAT le nom de l'état du processus et le symbole UNIQUE si le processus n'a aucun état défini dans la description Estelle.
- les variables du processus : pour chaque variable, on affiche son nom et sa valeur. Les variables de type autre que `boolean`, `char`, `integer`, `real` et `array [a..b] of char` sont affichées en hexadécimal).

**La liste des transitions tirables du processus** Pour chaque processus, s'affiche l'ensemble des transitions tirables qu'offre le simulateur à l'utilisateur. Pratiquement, on a un menu où figurent toutes les transitions tirables ainsi qu'une option *Néant*. On choisit la transition ou *Néant* par sélection avec la souris dans le menu. On peut passer d'un choix à l'autre pour visualiser les transitions tirables ; la transition n'est tirée qu'après avoir sélectionné le mode interactif dans la fenêtre de la simulation. Les transitions sont nommées par $t_{i,j}$ ou $t_i$ avec $i$ pour le numéro de la transition et $j$ pour le numéro d'indice de *any* lorsque celui-ci est différent de 0.

**Le texte de la transition couramment sélectionnée** Cette sous-fenêtre déroulante permet l'affichage du texte de la transition sélectionnée: cette transition ne sera effectivement tirée que lors de la sélection du bouton *interactif* dans la fenêtre de contrôle de la simulation. Le texte affiché correspond à celui de la description du protocole Estelle étudié.

### 3.4.7 La fenêtre d'historique

Les informations de cette fenêtre déroulante permettent de visualiser sous une forme synthétique le déroulement pas à pas de la simulation quelque soit le mode de simulation (y compris le mode *Rebrousser*). Les éléments représentés dans l'historique sont le numéro du PAS, le nom du PROCESSUS, la nom de la TRANSition tirée, l'ETAT DEPART, l'ETAT FINAL, et le ou les MESSAGE(S) ÉMI(S).

Ces données sont aussi sauvegardées à la fois dans un fichier texte de nom *nom-source.his*, et dans un fichier (avec un format interne) de nom *nom-source.int* qui permet de retourner à un pas antérieur de la simulation et de reprendre une nouvelle session à l'endroit où on s'était arrêté la fois précédente.

La gestion de ces fichiers est effectuée par l'utilisateur. Par exemple, le fichier *nom-source.his* n'est jamais détruit. Si l'utilisateur ne détruit pas le fichier *nom-source.int* après une simulation alors la simulation du même programme reprendra au même pas.

# Bibliographie


[1] A. Arnold. Construction et analyse des systèmes de transitions : le système MEC. In *Premier Colloque $C^3$, Angoulême, France*, A. Arnold, Septembre 1985.

[2] G. Boudol. *Le calcul MEIJE.* Editions du CNRS, 1985.

[3] S. Budkowski and P. Dembinski. An introduction to Estelle: a specification language for distributed systems. *Computer Networks and ISDN Systems*, 14:3-23, 1987.

[4] CCITT. *SDL, Recommendation Z.100.* 1987.

[5] J.-P. Courtiat, P. Dembinski, R. Groz, and C. Jard. Estelle : un langage ISO pour les algorithmes distribués et les protocoles. *Technique et Science Informatique*, 6(2), 1987.

[6] R. Groz. Vérification de propriétés logiques des protocoles et systèmes répartis par observation de simulations. *Thèse de doctorat de l'université de Rennes I, No 194*, January 1989.

[7] R. Guetschel. *Simulation interactive de programmes Estelle.* Rapport de Mastère, Ecole Supérieure d'Electricité, Rennes, à paraître, Septembre 1989.

[8] ISO. *LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* ISO/ DP 8807, March 1985.

[9] ISO 9074. *Estelle: a Formal Description Technique based on an Extented State Transition Model.* ISO TC97/SC21/WG6.1, 1989.

[10] ISO 9074. *Proposed draft addendum to ISO 9074:1989 – Estelle tutorial.* ISO 9074:1989 ISO/IECJTC1/SC21/F60.

[11] C. Jard, R. Groz, and J.F. Monin. Development of VEDA: a prototyping tool for distributed algorithms. In *IEEE Trans. on Software Engin.*, March 1988.

[12] C. Jard and J.-M. Jézéquel. *Un compilateur Estelle multiprocesseurs pour l'expérimentation d'algorithmes distribués sur machines parallèles.* Technical Report 453, IRISA University of Rennes, January 1989.

[13] C. Jard and M. Raynal. *Specification of Properties is Required to Verify Distributed Algorithms.* Technical Report 651, INRIA, Centre IRISA, Rennes, February 1987.

[14] J.-M. Jézéquel. Building a global time on parallel machines. In *Proc. of the 3rd International Workshop on Distributed Algorithms*, pages 136-147, Lecture Notes in Computer Science, Springer Verlag, 1989.

[15] J.-M. Jézéquel. Outils pour l'expérimentation d'algorithmes distribués sur machines parallèles. Thèse, Univ. Rennes I, Rennes, Octobre 1989.

[16] U. Johansen and E. Vefsnmo. Automatic program generation of SDL-specifications: principles and solutions. In *SDL'87, State of the Art and Future Trends*, North-Holland, 1987.

[17] J.L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in XÉSAR of the sliding window protocol. In $7^{th}$ *IFIP International Workshop on Protocol Specification, Testing and Verification, Zurich, Suisse*, North Holland, May 1987.

[18] S.M. Ulam. *The adventures of a mathematician*. Scribners, 1976.

[19] H. Zimmermann. OSI reference model of architectures for open systems interconnection. *IEEE Transactions on Communications*, COM-29, April 1980.

# Annexe A

# Notice d'utilisation succincte

La commande appelant le compilateur *ECHIDNA* est *estl*, sa syntaxe est proche de celle du compilateur *C* de Unix. La commande *estl* seule en donne la notice d'utilisation, qui est la suivante :

```
ESTL(1)                    *** ECHIDNA PROJECT ***                    ESTL(1)

NAME
  estl, an Estelle compiler for Distributed Computers: ECHIDNA project.

SYNOPSIS
  estl [-bcCtSGHIX] [-s sys] [-m file] [-l] [-o objFile] [estelleFile]

DESCRIPTION

The estl program  translates an Estelle  formal description  to C data
structures, and then compiles and  links them with a system  dependant
kernel, to produce code for  various architectures (see below).  If an
error is found in the source, your favourite editor (as defined in the
environment   variable  EDITOR, or  vi otherwise)   is  invoked on the
errored line.  When you leave it, compilation is resumed.
If no argument is given, the last version of this page is displayed.

OPTIONS
  -b  Batch mode : the editor is not invoked when an error is found.
  -c  Perform only Estelle to C translation.
  -C  Perform C compilation and link with the kernel.
  -t  Don't produce extra-code to check index range validity.
  -S  Include Estelle program symbol table in the code.
  -G  Debug option for use with dbx.
  -H  Establish a global time before an execution.
  -I  Interactive simulation using Suntools
  -X  Interactive simulation using X11 Motif (not yet available)
  -s  identifies the target system, which must be one of:
        unix    : a UNIX 4.2 system (default option)
        netsun  : a network of Sun (using TCP/IP)
        ipsc2   : an iPSC/2 (Intel Hypercube)
        fps     : an FPS-T40 (Transputer based FPS Hypercube)
        helios  : a T-node HELIOS (Transputer based Telmat machine)
        pc      : a PC (MS-DOS personnal computer with MSC)
```

```
  -m  Link with an extra object or C file module, endding with .o
      or .c (can be repeated)
  -l  Link with a library file
  -o objFile: Names the executable program "objFile".

USAGE
  You can provide the following optional parameters to your executable
  Estelle program:
    -dxxxx  The duration 'xxxx' of the execution in seconds
       (default=0 means infinity)
    -sxxxx A seed (integer) xxxx for the random generator
    -u  Prevent the bufferisation of trace output (default)
    -b  Forces the bufferisation of trace output
    -q  Make execution quiet: no header nor warning is displayed
    -a  Simulate an asynchronous execution of processes  within a node
    -D  Select a  deterministic semantic for the  execution: the first
        firable transition  of a process is actually fired, priorities
        are ignored.
    -T  Trace each interaction (Compile time option -S is needed).

  To load and run your program, see system specific instructions.
  For exemple:
  - On Unix machines and PC, enter:  program_name [arg1 ..[argn]]
  - On a Sun network:  getsun sun1 sun2 ..sunN
        where sun[1..N] are workstations actual symbolic names.
        Then you can either start "by hand" the Estelle program on
        each workstation (from the directory where you ran "getsun")
        as described above, or run the following command once
        to start the Estelle program on each machine (through rsh,
        so output is redirected to this workstation):
        loadsun "program_name [arg1 ..[argn]]"
  - On the ipsc/2:  getcube [cubesize] #see ipsc manual
        load program_name [arg1 ..[argn]]
  - On T-node (Helios) runec cubesize program_name [arg1 ..[argn]]

  If no execution duration is specified, a <CTRL C> must be entered
  to stop the program, and remaining processes should be killed.

ABOUT ESTELLE
  A  strict  subset of  the  standard  Estelle is  accepted  (the  main
  limitation is to force  the emptyness  of parent processes, the others
  are minor restrictions, e.g. not allowing embedding  of procedures  or
  transitions).    See the ISO  documents and published  papers for the
  language presentation.

  Echidna defines some predefined functions:
  - trace(format): like Pascal writeln
  - ntrace(format): like Pascal write
  - randomint(i1,i2): returns a randomly chosen integer between i1 an i2
                      (uniform distribution)
  - randomreal(r1,r2): the same but for real numbers
  - time: returns the local time of the processor being in charge of the
          Estelle process (in milliseconds since loading)
```

```
  - gtime: returns a global time and the associated precision; must be
           declared as primitive (assuming the H option).

  Estelle programs can be also interfaced with C-functions using Estelle
  primitives (see the generated C-code for interfacing data parameters).

BUGS
  Errored Estelle programs may not be detected (and possibly not detected
  by the C-compiler itself). This (rare) situation may appear in
  complex expressions involving different variable types.

  Bugs and others problems are to be reported to:

    J-M Jezequel TB105 IRISA poste 537 e-mail: jezequel@irisa.fr
  or:
    C. Jard      TB107 IRISA poste 538 e-mail: jard@irisa.fr

####### @(#) 9/6/91 --Echidna Package-- V3.1 (IRISA) #######
```

# Annexe B

# Messages d'erreurs du compilateur

**Erreurs :**

1. 'Array id expected'
2. 'Bad character'
3. 'Bad type for the id'
4. 'Boolean expression expected'
5. 'Bounds are of different types'
6. 'Call to proc/func/body expected'
7. 'End of file in comment'
8. 'End of file in string or character'
9. 'Expression must be of integer, real or set type'
10. 'Expressions are not assignment compatible'
11. 'Function assignement expected'
12. 'Guarded clauses cannot be duplicated'
13. 'Identical channel types required'
14. 'Identifier declared twice'
15. 'Identifier not declared'
16. 'Illegal clause in the initialisation part'
17. 'Incompatible channel roles'
18. 'Incorrect number of parameters'
19. 'Integer expression expected'
20. 'Label declared twice'
21. 'Label defined twice'
22. 'Label not allowed in a transition block'
23. 'Label not declared'

24. 'Naming allowed only for transition blocks'
25. 'Negative subrange length'
26. 'Newline in string or character'
27. 'Ordinal type expected'
28. 'Output not allowed on this port'
29. 'Scalar type definition expected'
30. 'Selection requires a record'
31. 'Value parameter expected'

**Restrictions d'implantation :**

1. '... has no meaning here'
2. 'Cannot handle dynamic statements'
3. 'Cannot handle nul-character in strings'
4. 'Cannot handle sets with base ¿¿ 0'
5. 'Cannot handle sets with very large range'
6. 'Const of any type has no meaning'
7. 'Delay clause not yet implemented in a DS'
8. 'Domain lists not allowed'
9. 'Embedding proc/func not yet allowed'
10. 'Embedding transitions is not yet permitted'
11. 'Exist function not implemented'
12. 'External bodies are not yet implemented'
13. 'Illegal construct for a static process'
14. 'Illegal construct for a static refinement'
15. 'Internal interaction points not allowed'
16. 'Memory overflow during parsing'
17. 'Non-deterministic initialization not allowed'
18. 'Only individual queues implemented'
19. 'Otherwise clause not implemented'
20. 'Sharing variables is not permitted'
21. 'Simple type expected'
22. 'Stateset definition not allowed'
23. 'Token buffer overflowed'
24. 'Too long identifier'
25. 'Too long string'
26. 'Too many machine integer types'
27. 'Too many strings, identifiers or real numbers'
28. 'With clause not implemented'

**Erreurs Internes :**

1. 'Bad name for machine integer type'
2. 'Bad tree structure'
3. 'Cannot find C-type for integer-subrange'
4. 'Cannot find tag'
5. 'Scope overflow'

**Avertissements :**

1. 'Extension: Output in subroutine'

# Annexe C

# Liste des fichiers utilisés par ECHIDNA

Voici une description succincte des différents fichiers de sources livrés avec *ECHIDNA* :

```
Makefile          Describes how to build and install ECHIDNA. See MAKE(1)
Manifest          This file
README            Installation description. Read this file first
bagrodia.stl      Estelle example source file: the Bagrodia algorithm
bitalt.stl        Estelle example source file: alternate bit protocol
commandes.uil     Part of the X11/Motif interactive interface
const_util.h      Some constants used with SunView interactive simulation
crible.stl        Estelle example source file: parallel Erathostene crible
echidnac.c        The Estelle-C compiler, translated from a Pascal source
echidnak.h        Some constants used with both generated code and kernel
estl.c            The Estelle compiler interface
estl.doc          The manual for the Estelle compiler (options and so on)
fen_sun_view.c    Primitives to handle windows within SunView
gencdl.c          Configures a T-node/Helios machine to run an Estelle program
getsun.csh        (C-shell) configures a sun network to run an Estelle program
gtime.c           Interface to "mes.c", the Global Time builder
historique.uil    Part of the X11/Motif interactive interface
interaction_i.c   Display interface for SunView interactive simulation
interaction_x.c   Display interface for X11/Motif interactive simulation
intersym.c        Interface to retrieve generated symbolic information
kernel.h          Some constants used only in the kernel
loadsun.csh       (C-shell) Runs an Estelle program across a sun network
mes.c             The Global Time builder, translated from an Estelle source
nom_fen_fran.h    French names for SunView window data types
parametres.uil    Part of the X11/Motif interactive interface
pec.c             Pretty prints an Estelle program using Latex format
process.uil       Part of the X11/Motif interactive interface
process_x.h       Constants used with X11/Motif interactive simulation
ra81.stl          Estelle example source file: Ricart & Agrawala algorithm
recapitulatif.uil       Part of the X11/Motif interactive interface
runec.csh         (C-shell) Runs an Estelle program on a T-node/Helios machine
runtime.c         The bulk of the Distributed Run-Time Kernel
scheduler.c       The various ECHIDNA schedulers, selectable at run-time
scheduler_i.c     The scheduler for SunView interactive simulation
```

```
scheduler_x.c   The scheduler for X11/Motif interactive simulation
session.uil     Part of the X11/Motif interactive interface
sys_fpsT20.c    A system interface for a FPS T20 (Transputer based)
sys_helios.c    A system interface for a T-node/Helios (Transputer based)
sys_ipsc2.c     A system interface for an iPSC/2 (i80386 based) hypercube
sys_netsun.c    A system interface for a network of Sun workstations
sys_pc.c        A system interface for a PC with MS/DOS
sys_unix.c      A system interface for a standart Unix system (both BSD&SYSV)
transition.uil  Part of the X11/Motif interactive interface
```

# Annexe D

# Installation de ECHIDNA

```
######### @(#) 9/6/91 --Echidna Package-- V3.1 (IRISA) ######################

Here is "estl" (the ECHIDNA compiler), its Distributed Run-Time Kernel
and some utilities.

******** USAGE IS RESTRICTED TO NON-COMMERCIAL PURPOSES *********

To install ECHIDNA, edit the Makefile to set system dependent
variables (be careful with unnecessary blank characters):
ECHIDNA: where to install object files and utilities
BIN    : where to install commands (a directory on the path)
SYS    : one of available: unix, netsun, ipsc2, fps, helios, pc
OPT    : Suntools and X11 interfaces: select any of: $(inters) $(interx)
SUFIX  :  depends on the system: o, to, OBJ (generally o )
CFLAGS : flags for C compiler, generally -O
CC     : name of local C compiler, generally cc

Example for a standard Sun 3 or SUN 4 installation:
#####################################################################
#######these macros should be set here or on command line#########
#######    Be careful with blank characters              ########
# where to install object files and utilities,
# also value of ECHIDNA environment variable
ECHIDNA =/usr/local/echidna
# where to install commands (a directory on the path)
BIN =/usr/local/bin
# one of available: unix, netsun, ipsc2, fps, helios, pc
SYS =netsun
# Suntools and X11 interfaces: select any of: $(inters) $(interx)
OPT =$(inters)
# depends on the system: "o", "to", "OBJ"
SUFIX =o
CFLAGS = -O
CC = cc
#####################################################################

Then enter the following command:
> make -f Makefile install
```

In order to use actually ECHIDNA, one must set the environment variable ECHIDNA to the directory where object files and utilities are installed.
Furthermore, if you want to use the X11/Motif interface, you have to set the environment variables UIDPATH to $ECHIDNA/%U and LIBXESTL to -L/usr/local/motif/lib -lMrm -lUil -lXm -L/usr/lib/X11 -lXt -lXmu -lX11 -lXext (However, this may vary according to your X11/Motif installation)

- The manual for the estl compiler is available in file estl.doc, or hitting estl without parameters.
- You can find various examples of Estelle programs in this directory (.stl files)
- pec is a command to pretty print an Estelle program using Latex format: usage: pec [-i] < foo.stl > foo.tex; latex foo.tex
  Option -i makes pec produce a file to be included in a Latex document.

Remember: this is a prototype implementation of Estelle for distributed computers. Its main restrictions are:

- Strict subset of IS9074, but without dynamic constructs: empty parents.

- Delay clause non implemented (meaningless in distributed environment)

- Nesting of procedures is not allowed.

- Module instances must be initialized before their ips can be connected.

Any comment or problem: mail to jezequel@irisa.fr

# Index

## LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 593 APPLICATION OF BELLEN'S PARALLEL METHOD TO ODE's WITH DISSIPATIVE RIGHT-HAND SIDE
Philippe CHARTIER
Juin 1991, 24 pages.

PI 594 PROGRAMMATION D'UN NOYAU UNIX EN GAMMA
Pascale LE CERTEN, Hector RUIZ BARRADAS
Juillet 1991, 48 pages.

PI 595 CALCULATING THE BUSY PERIOD DISTRIBUTION OF THE M/M/1 QUEUE
Louis-Marie LE NY, Gerardo RUBINO, Bruno SERICOLA
Juillet 1991, 11 pages.

PI 596 EFFICIENT CODE GENERATION FOR DISTRIBUTED MEMORY MACHINES
Françoise ANDRE, Olivier CHERON, Jean-Louis PAZAT, Henry THOMAS
Juillet 1991, 14 pages.

PI 597 KOAN : A SHARED VIRTUAL MEMORY FOR THE iPSC/2 HYPERCUBE
Zakaria LAHJOMRI, Thierry PRIOL
Juillet 1991, 32 pages.

PI 598 KOAN : A VERSATILE TOOL FOR PARALLELIZING REALISTIC RENDERING ALGORITHMS
Didier BADOUEL, Kadi BOUATOUCH, Zakaria LAHJOMRI, Thierry PRIOL
Juillet 1991, 28 pages.

PI 599 STATISTICAL ESTIMATION OF ROUNDOFF ERRORS AND CONDITION NUMBERS
Jocelyne ERHEL
Septembre 1991, 34 pages.

PI 600 NOMBRE DE SOLUTIONS ET SATISFIABILITE D'UN PROBLEME SAT ; UNE APPROCHE ENSEMBLISTE, COMBINATOIRE ET STATISTIQUE
Israël-César LERMAN
Septembre 1991, 88 pages.

PI 601 ACCELERATED STOCHASTIC APPROXIMATION
Bernard DELYON, Anatoli JUDITSKY
Septembre 1991, 22 pages.

PI 602 MINIMUM VARIANCE CONTROL : RANDOM AUTOREGRESSIVE PARAMETERS
Anatoli JUDITSKY
Septembre 1991, 26 pages.

PI 603 L'EXPERIMENTATION D'ALGORITHMES DISTRIBUES SUR MACHINES PARALLELES AVEC ECHIDNA
Jean-Marc JEZEQUEL
Claude JARD
Septembre 1991, 64 pages.



Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique.

ISSN 0249-6399